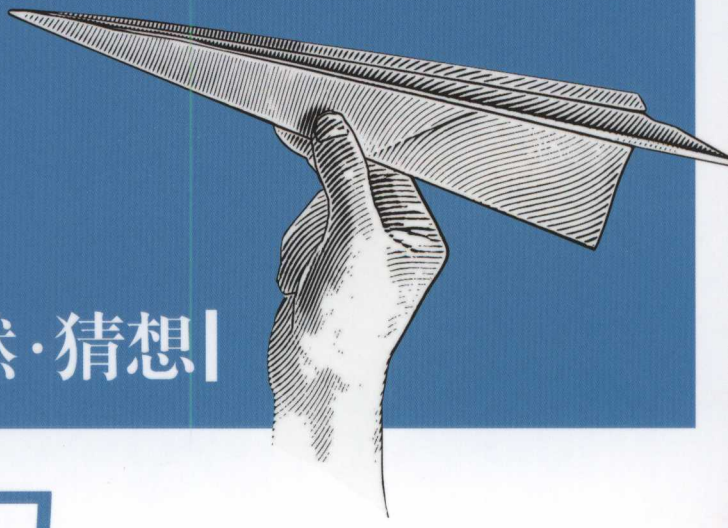
追求简单·自然·猜想



图解算法

发现算法
理解算法
主动猜算法

俞征武 著

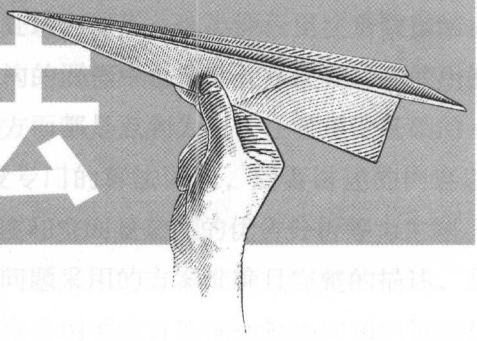
从对话讨论中自然地引出算法的意义

看图理解算法基础，轻松打开算法之门

培养“猜”的习惯，主动思考，解决面临的编程问题

机械工业出版社
China Machine Press

图解算法



俞征武 著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

图解算法 / 俞征武著 .—北京：机械工业出版社 ,2017.9

ISBN 978-7-111-57887-1

Ⅰ . ①图… Ⅱ . ①俞… Ⅲ . ①计算机算法 – 图解 Ⅳ . ① TP301.6-64

中国版本图书馆 CIP 数据核字（2017）第 216827 号

算法是人们利用电脑解决问题的技巧。本书以轻松的对话方式，采用图解的辅助说明，帮助读者简单、自然地掌握算法的基本概念，并养成主动思考的习惯，达到用算法解决实际问题的目的。

全书共分 12 章，内容包括一切从观察开始、分而治之法、动态规划、贪婪法、修剪与搜索法、树搜索法、问题转换、图算法、计算几何、算法的难题、逼近算法、随机算法等。

本书示例丰富，图文并茂，以易于理解的方式阐释算法，帮助程序员在日常项目开发中更好地发挥算法的能量。

图解算法

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）

责任编辑：夏非彼　迟振春　　责任校对：王　叶

印　　刷：中国电影出版社印刷厂　　版　　次：2017 年 9 月第 1 版第 1 次印刷

开　　本：180mm × 230mm　1/16　　印　　张：17.25

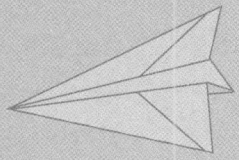
书　　号：ISBN 978-7-111-57887-1　　定　　价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426　88361066　　投稿热线：（010）88379604

购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

推荐序



接受过计算机软件专业系统学习的人应该都学过“数据结构”“数据结构与算法”一类的课程，这类课程在算法的讲授上或浅或深，多数是以数据结构为核心讲述算法的具体运用和实现的。这类数据结构课程的主要内容是以计算机数据的存储结构为基础，将需要处理的现实世界的信息或者数据之间存在的一种或多种特定关系抽象化为计算机中可以描述的数据元素的集合，再辅以高效的检索算法和索引技术进行处理和加工。这类课程更注重数据结构的部分，虽然理工类大学在这类数据结构的课程中也会涉及算法设计和使用的时间、空间、效率等内容，但是在算法方面都是点到为止。注重理论基础的一些大学除了数据结构课程外，还会开设专门的算法课程，后者讲述的内容以算法的设计、研究以及算法的时间复杂度和空间复杂度的优劣分析等为主题。

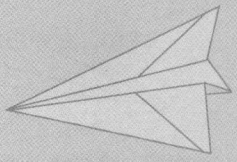
算法的核心是对要解决的问题采用的方案准确且完整的描述，是解决问题的一系列连续、清晰的指令，也是用系统方法描述解决问题的策略机制。对于同一类问题，不同的算法在时间、空间和效率上可能差别很大，它们的优劣可以用科学的时间复杂度与空间复杂度来衡量。

本书虽然不能算得上算法理论研究内容的鸿篇巨著，但是可以作为学习数据结构这类课程的进修教材，或者作为学习算法艰深理论课程的入门读本和参考指南。本书包含算法基本概念的脉络和算法设计的朴素思想，阐述了算法设计和分析的任督二脉，让读者在简单、自然的氛围中打通“烧脑”的算法世界。

资深架构师 赵军

2017年3月

前言



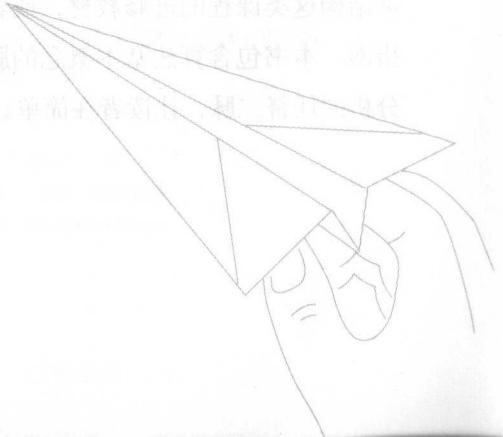
编写这本书的动机是希望帮助读者简单、轻松地掌握算法的基本概念。因此，本书将不尝试收录所有的算法，同时也不把有限的笔墨用来分析算法的复杂度和对算法进行严格证明。

本书在介绍算法之前，常常会刻意地加入一小段对话，目的是希望通过思辨和讨论，自然地引出算法的直观意义。倘若读者从学习中顺便养成思考的习惯，那就更好了。

作者知识面有限，再加上表达能力不足，如果导致书中仍有无法被读者理解之处，在此向读者致歉。假如您在阅读的过程中惊讶地发现算法之美，在此表示深深的敬意。

作者

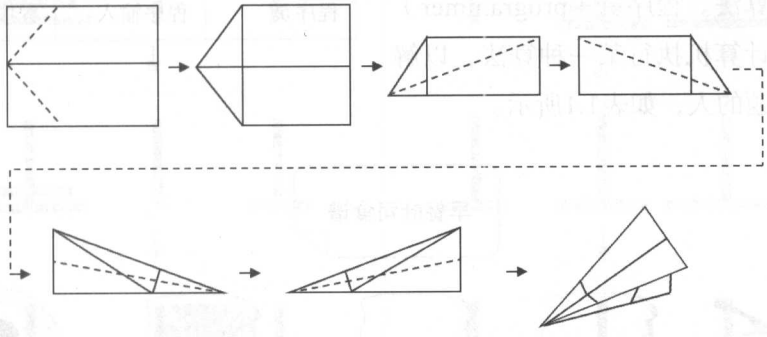
2017 年 4 月



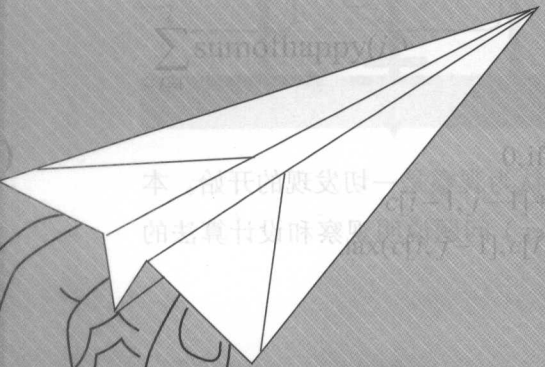
目 录

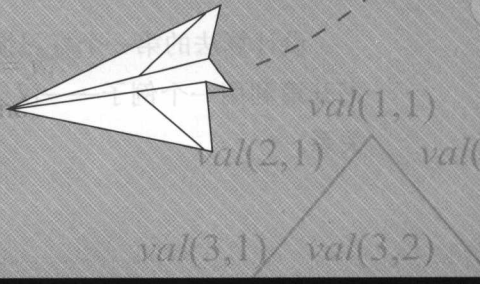
第1章 一切从观察开始

章节大纲



折纸飞机的过程隐含一个算法





1.1 什么是算法

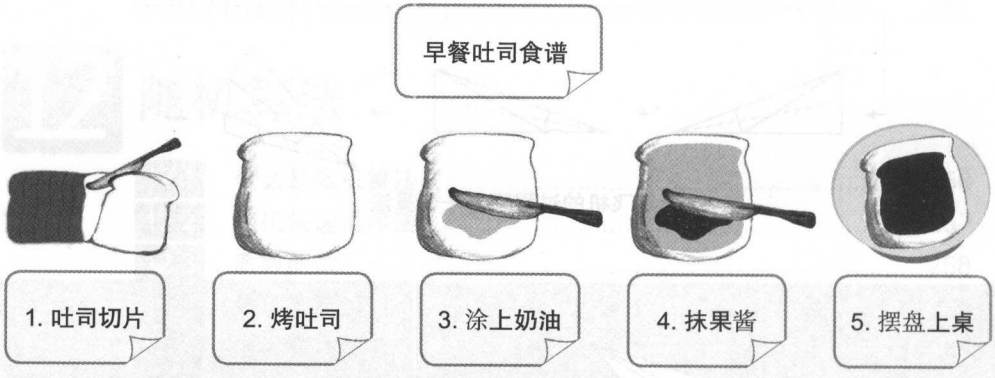
什么是算法（algorithms）？

简而言之，算法是在符合问题的限制下，将输入（input）转换成输出（output）的过程。

计算机算法是人类利用计算机解决问题的技巧之一。其实每个人都会一些算法。例如，厨师会将食材转换成美食；演奏家会将乐谱转换成迷人的音乐；小朋友将废纸折成纸飞机的过程也隐含一种算法。程序员（programmer）就是使用计算机执行某一种算法，以解决特定问题的人，如表1.1所示。

表 1.1　算法存在于各行业中

执行者	输入	输出
厨师	食材	美食
演奏家	乐谱	音乐
小朋友	废纸	纸飞机
程序员	程序输入	程序输出



如何设计算法?

设计算法的第一个好习惯就是观察。我们认为观察是一切发现的开始。本书开篇利用一个例子——汉诺塔（Hanoi Tower）问题说明观察和设计算法的关系。

1.2　汉诺塔问题

汉诺塔问题是一个古老的游戏。游戏的目的是将左方柱子上的盘子搬到右方的柱子。游戏的规则有三条：

（1）一次搬一只盘子。

（2）每根柱子只有最上面的盘子可被搬动。

（3）大盘子不可置于小盘子的上方。

图1.1所示为三只盘子汉诺塔问题的一个解法。

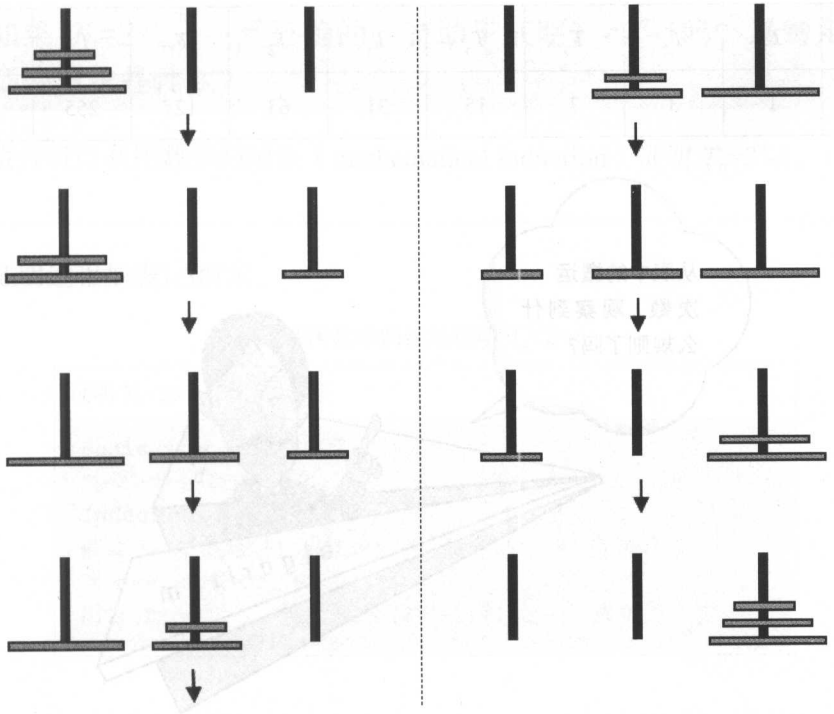


图 1.1　三只盘子汉诺塔问题的一个解法

从上图得知，三只盘子的汉诺塔问题共需要7个步骤完成。很自然地，我们提出第一个问题：

n只盘子的汉诺塔问题，共需要几个步骤完成？

若这个问题无法立刻回答，则可先观察一些范例。

观察一些小例子，并记录其移动的次数。

为记录方便，我们使用数学符号T_n代表n只盘子汉诺塔问题所需要的搬运次数。尝试算出n小于9之前的T_n，如表1.2所示。

表 1.2 观察汉诺塔问题的搬运次数

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
0	1	3	7	15	31	63	127	255	?

这个数列有什么规则？T_9 =?

T_n好像是很靠近 2 的几次方，T_n是2^n -1 吗？

如何证明 T_n 是 2^n-1?

不知道。

还有其他规则吗?

比较前后项的关系，好像后项是前项的两倍？不，是两倍加 1。

可以使用符号将此关系写下来吗?

可以，$T_n=2T_{n-1}+1$。

为何前后项有这样的关系?

不知道。

如果 $T_n=2T_{n-1}+1$ 是正确的，有助于证明$T_n=2^n-1$吗？最常用的证明方法是什么?

也许可以利用数学归纳法（mathematical induction）证明 $T_n=2^n-1$。

证明过程如表1.3所示。

表 1.3 利用数学归纳法证明 $T_n=2^n-1$

证明 $T_n=2^n-1$，当 $n\geqslant 0$ 时
Basis step: $T_0=2^0-1=1-1=0$，成立。 **Inductive step:** 假设 $n<k$，$T_n=2^n-1$ 成立。 当 $n=k$，$T_k=2\times T_{k-1}+1$。 因为 $T_{k-1}=2^{k-1}-1$，故 $T_k=2\times(2^{k-1}-1)+1=2^k-1$，成立。

另一个简单的证明如表1.4所示。

表 1.4 $T_n=2^n-1$ 的另一个证明

证明 $T_n=2^n-1$，当 $n\geqslant 0$ 时
$T_0=0$ $T_n=2T_{n-1}+1$ $T_n+1=2T_{n-1}+2$ 令 $U_n=T_n+1$，则 $U_0=1$，$U_n=2U_{n-1}$，$U_n=2^n$ $T_n=U_n-1=2^n-1$。

我们只利用简单的观察便猜中了 $T_n=2^n-1$ 这个性质。虽然不知道为什么 $T_n=2T_{n-1}+1$，但此性质有助于证明 $T_n=2^n-1$。接下来的一个问题是：

给出一个有n只盘子的汉诺塔问题，如何（找出）搬动盘子（的算法）？

若这个问题无法立刻回答，则可先尝试找出一个小例子的解答，并观察这个解答的规则。

以下为有 4 只盘子的汉诺塔问题的一个解答。根据先前发现的性质，我们得知共需要搬动 $2^4-1=15$ 次，如图1.2所示。

搬动的过程中有什么规则呢？

好像有些成堆的盘子打散之后又成堆了。

可以解释其中的道理吗？

大概是为了搬动最底部且最大的盘子，必须先将上面的3只盘子全部搬到中间的柱子。

哪个步骤可以看出此特性？

步骤 7。

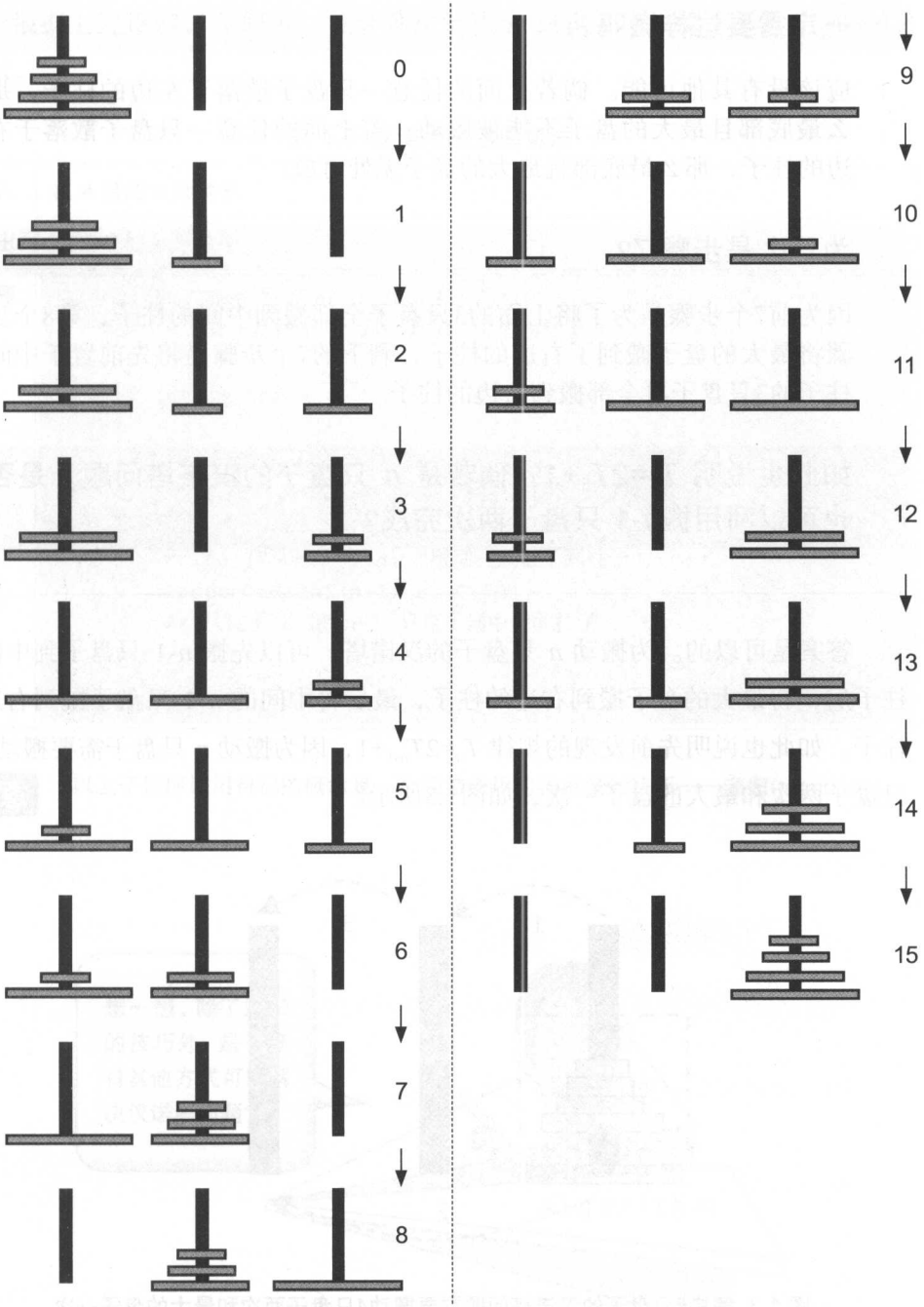


图 1.2　有4只盘子的汉诺塔问题的搬动过程

一定需要这样搬吗?

应该没有其他可能。倘若上面的任意一只盘子散落于左边的柱子，那么最底部且最大的盘子不能被搬动；若上面的任意一只盘子散落于右边的柱子，那么最底部且最大的盘子无处可放。”

为什么是步骤 7?

因为前7个步骤是为了将上面的3只盘子全部搬到中间的柱子，第8个步骤将最大的盘子搬到了右边的柱子，剩下的7个步骤是将先前置于中间柱子的3只盘子再全部搬到右边的柱子。”

如此也说明 $T_4=2T_3+1$? 倘若是 n 只盘子的汉诺塔问题，是否也可以利用搬n-1 只盘子两次完成?

答案是可以的。为搬动 n 只盘子的汉诺塔，可以先搬 n-1 只盘子到中间的柱子后，将最大的盘子搬到右边的柱子，最后将中间的 n-1 只盘子搬到右边的柱子。如此也说明先前发现的规律 $T_n=2T_{n-1}+1$，因为搬动 n 只盘子需要搬动 n-1 只盘子两次和最大的盘子一次，如图1.3所示。

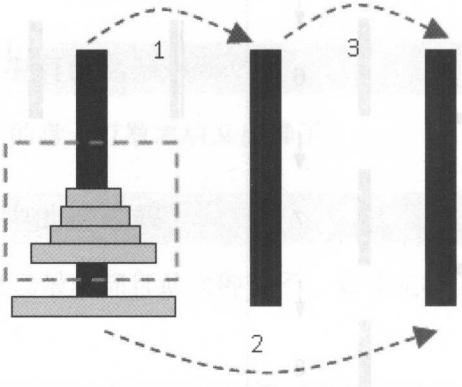


图 1.3 解决5只盘子的汉诺塔问题需要搬动4只盘子两次和最大的盘子一次

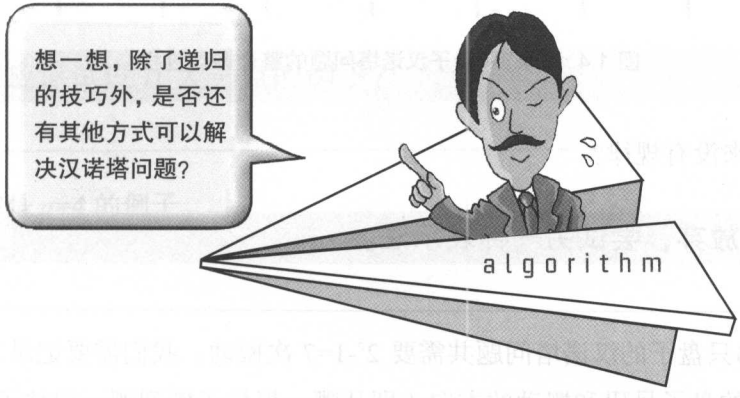
根据上述道理，我们可以设计算法解决 n 只盘子的汉诺塔问题，如表1.5所示。

表 1.5 汉诺塔的递归算法

输入	在 A 柱的 n 只盘子
输出	在 C 柱的 n 只盘子
步骤	

```
Algorithm TowersOfHanoi (n, A, C, B)
/* 搬动 n 只盘子从柱子 A 到柱子 C，可借用柱子 B*/
{
    if (n>=1) then
    {
        TowersOfHanoi(n-1, A, B, C);
        /* 从柱子 A 搬 n-1 只盘子到中间的柱子 B  */

        write ("从柱子", A, "搬动一只盘子到柱子", C);
        TowersOfHanoi(n-1, B, C, A);
        /* 从柱子 B 搬 n-1 只盘子到中间的柱子 C */
    }
}
```

注意　以上子程序调用自己的概念是一个经常使用的程序设计技巧——递归(recursion)。



1.3　汉诺塔问题的非递归算法

我们已经找到一个解决汉诺塔问题的递归算法，另一个有趣的问题是：

如果不用递归算法，可以解决汉诺塔问题吗？

好像不行？

没关系，先观察几个小例子。注意从输入转换成输出的过程，并且用符号记录整个移动过程。

3只盘子汉诺塔问题的搬动过程如图1.4所示。为了方便，利用1到3 的整数代表从小到大的3只盘子，而 ABC 代表3根柱子的位置。

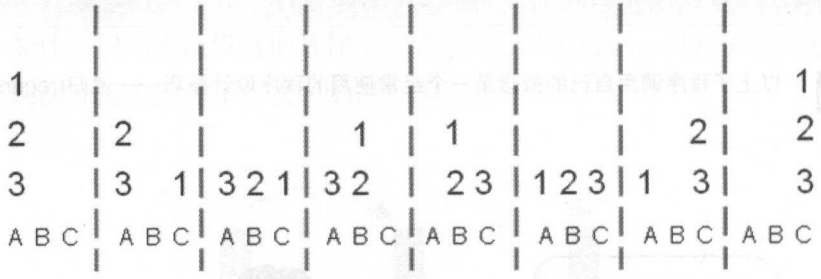


图 1.4　记录3只盘子汉诺塔问题的整个搬动过程

看起来没有规律？

不要放弃，尝试另一种表示法。

因为3只盘子的汉诺塔问题共需要 2^3-1=7 次搬动。我们需要记录7次搬动中每次搬动的盘子号码和搬动的方向（即从哪一根柱子搬到哪一根柱子），如图1.5所示。

步骤	1	2	3	4	5	6	7
移动的盘子	1	2	1	3	1	2	1
移动的方向	A→C	A→B	C→B	A→C	B→A	B→C	A→C

图 1.5　换一种方式记录3只盘子的汉诺塔搬动过程

还是看不出有什么规律?

不要放弃，再尝试另一种表示法。

因为搬动的方向似乎不易观察出规律，所以图1.6将移动的方向标注得更简洁一些。

步骤	1	2	3	4	5	6	7
移动的盘子	1	2	1	3	1	2	1
C	↑		↓	↑		↑	↑
B		↑			↓		
A							

图 1.6　换一种方式记录3只盘子的汉诺塔搬动过程

也许应该试试 *n* 大一点的例子?

图1.7是 n=4 的例子。

步骤	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
移动的盘子	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
C		↑	↑		↓	↓		↑	↑		↓	↑		↑	↑
B	↑	↑	↑	↑	↓	↓	↑	↑	↑	↓	↓	↑	↑	↑	↑
A	↑	↑		↑	↓		↑	↑		↓	↓		↑	↑	

图 1.7　记录4只盘子的汉诺塔搬动过程

注视搬运的盘子号码，是否找得到规律?

好像1号盘子出现在奇数的搬动次数上?

其他盘子也有这种现象吗?

2 号盘子没有这个现象。但是，好像每个 2 号盘子隔4次才搬动一次?

专心观察2号盘子被搬动的步骤，如图1.8所示。

步骤	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
移动的盘子		2				2				2				2	

图 1.8　观察 2 号盘子被搬动的步骤

还有其他盘子有类似的现象吗?

3 号盘子隔8次出现一次。

所有盘子有一致的规律吗?

啊，我知道了。k 号盘子每隔 2^k 次搬动一次。

知道第 i 步骤搬动第几号盘子吗?

不知道，但是第 1，3，5，7，9，11，13，15 步是搬动第1号盘子，而

第 2，6，10，14 步是搬动第2号盘子，第4和12步是搬动第3号盘子，而第8步搬动第4号盘子。

耶，我知道了。可能和这些数字的因子为2的倍数有关。

可以用符号写下来吗？

第 i 步搬动盘子的号码与 i 最大的因子 2^k 有关。

我们终于知道，当进行第 i 步时，找出最大的 k 使得 $i=2^k \times z$（z 为正整数）。因此，在第 i 步时，需要搬动第 $k+1$ 号盘子。

每次盘子搬动的方向知道了吗？

也许我们该看看，1 号盘子搬动方向的规律是什么？

图 1.9 标注了 1 号盘子在整个搬动过程中的搬动方向。

步骤	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
移动的盘子	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
C			↑		↓				↑		↓				↑
B	↑		↑		↓		↑		↑		↓		↑		↑
A	↑				↓		↑				↓		↑		

图 1.9 观察 1 号盘子的搬动方向

有规律吗？

先看看1号盘子搬动的方向，好像往上移动到顶后就掉下来？

再看看2号盘子搬动的方向。有规律吗？

1号盘子和2号盘子搬动的方向有一些相像，但又不一样。

3 号呢?

啊，我知道了。3号盘子和1号盘子搬运的方式一致，也就是奇数号盘子搬动的方式一致。

偶数号盘子搬动的方式一致?

是的，偶数号盘子搬动的方式一致，只是方向正好相反。也就是往下移动到底后，就跳上去。

图1.10 标注了偶数号盘子在整个搬动过程中的搬动方向。

步骤	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
移动的盘子	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
C		↑				↓		↑						↑	
B		↑				↓		↑		↓				↑	
A		↑						↑		↓				↑	

图 1.10 观察偶数号盘子的搬动方向

试一试其他例子，看看是否仍保留此规律呢?

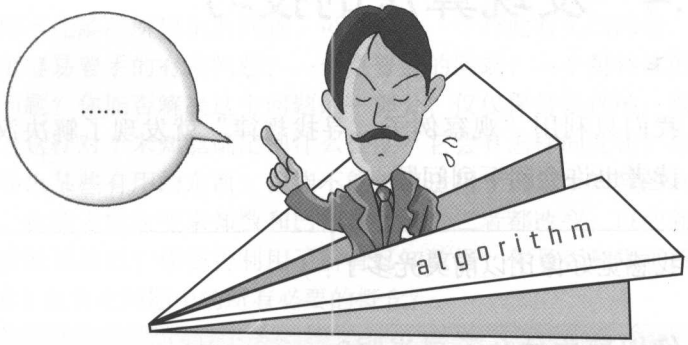
当汉诺塔问题的盘子数是奇数时，好像整个搬动的方向全部相反。

可以换个方式描述您的观察吗?

当汉诺塔问题的盘子数是奇数时，奇数号盘子往下移动到底后，就跳上去；而偶数号盘子往上移动到顶后，就掉下来。

这些性质如何证明?

前面不是说，这本书不会提及证明的吗?



根据以上讨论，可设计一个解决汉诺塔问题的非递归算法，如表1.6所示。为了方便起见，将3根柱子编号命名为0（起始的柱子）、1（可暂放的柱子）、2（目标的柱子）。

表 1.6 汉诺塔问题的非递归算法

输入	汉诺塔的盘子数为 n，盘子编号为{1，2，…，n}。3根柱子编号为 0、1、2
输出	全部 2^n-1 次移动过程
步骤	一个循环控制整数 i 从 1 到 2^n-1 执行下列步骤完成每一只盘子的移动 Step 1: 计算最大的 k，使得 $i=2^k \times y$(故本次将搬动第 k+1 号盘子) Step 2: 当 n 为偶数且 k+1 为奇数时，移动第 k+1 号盘子从当前的 s 柱子搬到第(s+1) MOD 3 柱子 当 n 为偶数且 k+1 为偶数时，移动第 k+1 号盘子从当前的 s 柱子搬到第(s-1) MOD 3 柱子 当 n 为奇数且 k+1 为奇数时，移动第 k+1 号盘子从第 s 柱子搬到第(s-1) MOD 3 柱子 当 n 为奇数且 k+1 为偶数时，移动第 k+1 号盘子从第 s 柱子搬到第(s+1) MOD 3 柱子

1.4　发现算法的技巧

我们只利用“观察例子，寻找规律”就发现了解决汉诺塔问题的两个算法，读者也许会问下列问题：

我感觉好像比以前灵光多了？

你以前为什么不灵光呢？

以前我只会努力记住别人教过的问题和方法。

现在不同了吗？

有一点感觉，但是怎样才有机会发现更多算法呢？

也许，波利亚先生（G. Polya）在《怎样解题》（How to Solve It）中的一段话，可回答发现算法的技巧。

怎样解题 (How to Solve It)

（节录并翻译自波利亚的《怎样解题》一书）

首先，你必须弄清问题。

未知数是什么？已知数据是什么？条件是什么？满足条件是否可能？要确定未知数，条件是否充分？或者它是否不充分？或者是多余的？或者是矛盾的？画张图，引入适当的符号，把条件的各个部分分开，你能否把它们写下来？

其次，找出已知数与未知数之间的联系。如果找不出直接的联系，你可能不得不考虑辅助问题。你应该最终得出一个求解的计划。

你以前见过这个问题吗？你是否见过相同的问题而形式稍有不同？你是否知道与此有关的问题？你是否知道一个可能用得上的定理？看着未知数！试想出一个具有相同未知数或相似未知数的熟悉的问题。这里有一个与你现在的问题有关，且早已解决的问题，你能应用它吗?你能不能利用它？你能利用它的结果吗？为了能利用它，你是否应该引入某些辅助元素？你能不能重新叙述这个问题？你能不能用不同的方法重新叙述它？

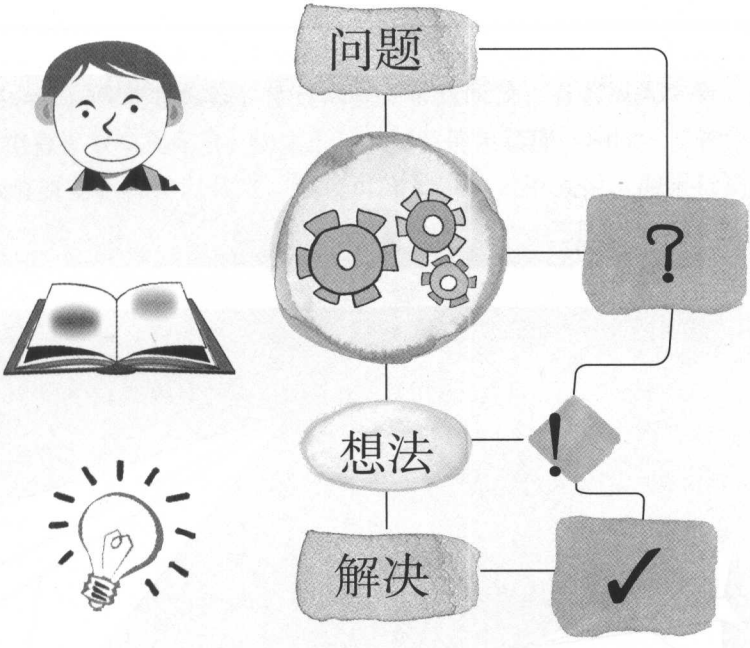
回到定义去。如果你不能解决所提出的问题，可先解决一个与此有关的问题。你能不能想出一个更容易着手的有关问题？一个更普遍的问题？一个更特殊的问题？一个类比的问题？你能否解决这个问题的一部分？仅仅保持条件的一部分而舍去其余部分，这样对于未知能确定到什么程度？它会有怎样的变化？你能不能从已知数据导出某些有用的东西？你能不能想出适用于确定未知数的其他数据？如果需要，你能不能改变未知数和已知数，或者二者都改变，以使新未知数和新已知数彼此更接近？你是否利用了所有已知数据？你是否利用了所有条件？你是否考虑了包含在问题中的所有必要的概念？

再次，实行你的计划。

实现你的求解计划，检验每一个步骤。你能否清楚地看出这一步是正确的？你能否证明这一步是正确的？

最后，验算所得到的解。

你能否检验这个论证？你能否用别的方法导出这个结果？你能否一下子看出它来？你能不能把这结果或方法用于其他问题？



学习效果评测

1. 编写一个递归的程序和一个非递归的程序解决汉诺塔问题，比较两者所需的运行时间，并写下你的观察与想法。

2. 老王开杂货店想送 N 块冬瓜糖砖给客户，每块冬瓜糖砖长、宽、高都是10厘米。老王希望将这 N 块冬瓜糖砖包装成一大包（x×y×z 的长方体），以方便运送，但为了响应环保，希望使用的包装纸越少越好。编写一个程序输入N，输出最少的包装纸面积。

输入：
9

输出：
3000

3. 编写一个程序，对一个正整数进行质因子的分解。例如，$3080=2^3\times5\times7\times11$。

输入：
3080

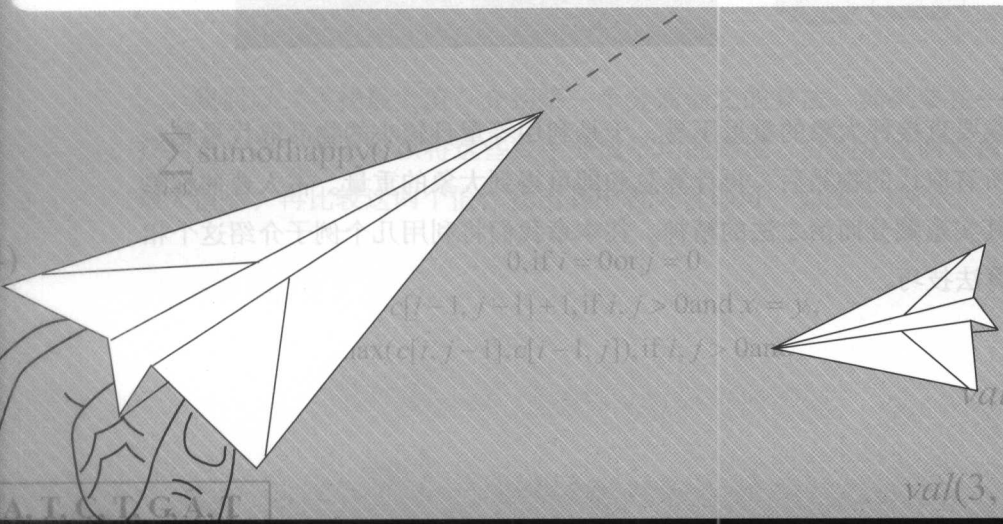
输出：
2^3 ×5×7×11

分而治之法

章节大纲

“邓哀王冲，字仓舒。少聪察岐嶷，生五六岁，智意所及，有若成人之智。时孙权曾致巨象，太祖曹操欲知其斤重，访之群下，咸莫能出其理。冲曰：‘置象大船之上，而刻其水痕所至，称物以载之，则校可知矣。’太祖大悦，即施行焉。”

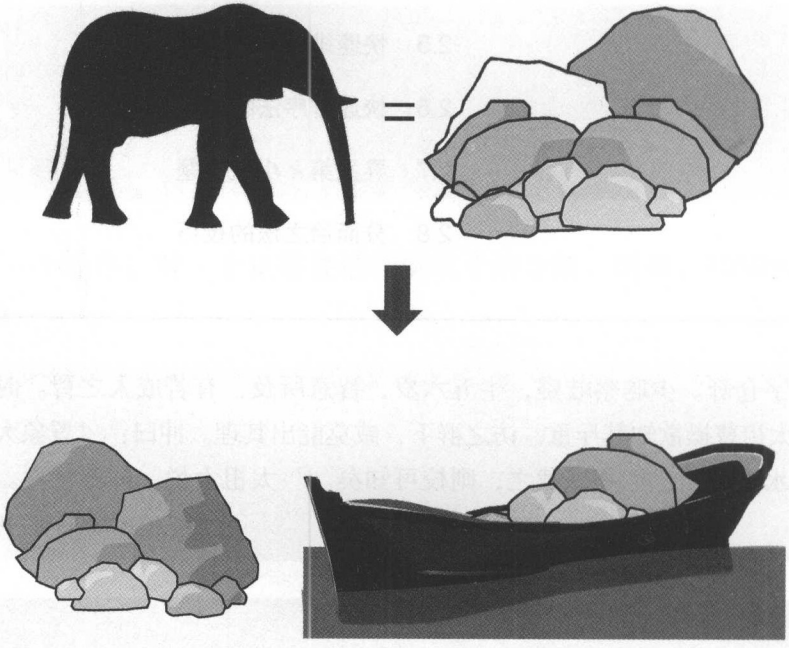
三国志・魏书曹冲传



2.1　何谓分而治之法

何谓分而治之法（divide and conquer）？

简而言之，就是将一个问题分割（divide）成一些小问题，并且递归地解决（conquer）后，再利用这些小问题的解合并（merge）成原来大问题的解。”



古时候要直接秤大象的重量不易，于是利用等重且较小的物品取代量测，分次称出所有取代的物品后，再计算总和即可得到大象的重量。古人曹冲称象的典故，其实蕴藏分而治之法的精神。在本章我们将利用几个例子介绍这个相当常见的算法技巧。

2.2　找出最大值

第一个问题是在一个整数的集合中找到最大值，如表2.1所示。

表 2.1　找出最大值

问题	在 n个数字中找出最大值
输入	任意的 n个数字不规律地存储于 A 矩阵中 4、2、17、5、22、8、13、6
输出	此 n 个数字中最大值为22

寻找最大值（finding the maximum）使用一个循环便可轻易解决，如表2.2中的算法所示。

表 2.2　寻找最大值的算法

输入	n个数字存储于 A [1:n]中
输出	max : n 个数字中的最大值

步骤	Step 1: 设置第一个元素为最大值，即 max = A[1]。
	Step 2: 执行循环 For i=2 to n do
	{
	If (A[i] > max) then max = A[i]。
	}
	Step 3: 输出 max。

我们以“寻找最大值”介绍第一个分而治之的算法。如果要在一堆数字中找到最大值，那么可以将这些数字分成平均的两堆。分别从这两堆数字中找到最大值后，再比较这两个值，找出其中较大者即可，如图 2.1所示。

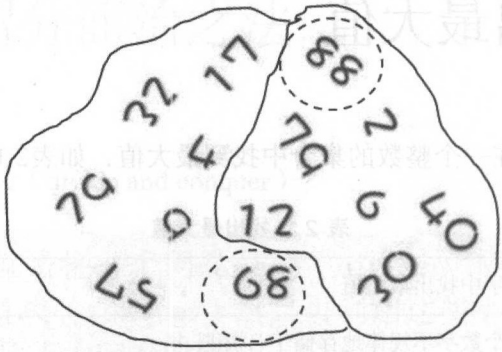


图 2.1 将一堆整数分成两堆后，再来找最大值

若将 100 个数据存放在矩阵 $A[1:100]$ 中，首先将此矩阵从中间分成两个均等的小矩阵 $A[1:50]$ 和 $A[51:100]$（此为分割（divide）步骤）。若我们可以在 $A[1:50]$ 中找到最大值，而且在 $A[51:100]$ 中找到另一个最大值，则从这两数中找出更大的值（此为 conquer 步骤），即可得到答案。利用这样的概念，第一个分而治之的算法就可以被设计出来了，如表2.3所示。

表 2.3 寻找最大值的分而治之算法

输入	*n* 个数字存储于 $A[i:j]$
输出	*Vmax*:*n* 个数字中的最大值

步骤	Procedure max-divide-and-conquer (*i*, *j*, *Vmax*)
	{
	Step 1: if *i*=*j* then *Vmax*=*A*(*i*); 并输出 *Vmax* ; break。
	Step 2: if *i*=*j*-1 then *Vmax*=*max*(*A*[*i*], *A*[*j*]); 并输出 *Vmax* ; break。
	Step 3: 计算矩阵中间位置; 即 *mid*=(*i*+*j*)/2。
	Step 4: 递归调用找出一半矩阵中的最大值; 即 max-divide-and-conquer(*i*, *mid*, *Vmax1*)。
	Step 5: 递归调用找出另一半矩阵中的最大值; 即 max-divide-and-conquer(*mid*+1, *j*, *Vmax2*)。
	Step 6: 输出 *Vmax1*和*Vmax2* 中更大的值; 即 *Vmax*=*max*(*Vmax1*, *Vmax2*), 输出 *Vmax*。
	}

2.3　时间复杂度

表2.2和表2.3中的两个算法哪一个比较好？

怎样评判一个算法的好与坏？

比如拥有较少的程序运行时间，也就是较小的时间复杂度（time complexity）。

什么是算法的时间复杂度？

简而言之，就是一个算法要花多少时间执行。

以找出最大值的算法（见表 2.2）为例，每条指令被执行的次数如表2.4所示。

表 2.4　寻找最大值的算法时间复杂度

寻找最大值算法的指令	执行次数
max=*A*(1)	1
For *i*=2 to *n* do	*n*
If *A*(*i*)>*max* then *max*= *A*(*i*)	*n*-1
Output *max*	1
执行指令总数	2*n*+1

这个算法的时间复杂度为 2*n*+1，此 *n* 是输入数值的个数。因为 2*n*+1 是 *n* 的两倍加1，当 *n* 够大时，2*n* 和 3*n*、5*n* 差不多和 *n* 成正比，因此我们常用 *O*(*n*)（读作 big O n）简称算法的时间复杂度。表2.5将介绍 *O* 的正式定义。

表 2.5 O 的正式定义

符号	O
定义	$O(g(n))$代表一个函数的集合{$f(n)$:存在一个正常数 c 和 n_0，使得所有的 $n \geq n_0$ 时，$0 \leq f(n) \leq cg(n)$}
范例	$3n=O(n)$ $100n=O(n)$ $300n+100=O(n)$ $300n^2-40n+50=O(n^2)$
使用时机	利用 $O(g(n))$将算法分成不同的层级

根据 O 的定义，表 2.2 上的寻找最大值的算法有 $2n+1=O(n)$的时间复杂度。相对地，寻找最大值的分而治之算法（表2.3）的时间复杂度为多少呢？执行 Procedure max-divide-and-conquer(1, n, $Vmax$)所需的计算时间为 $T(n)$。因为此算法将输入的矩阵平均分割后，调用自己两次，所以我们可以将 $T(n)$用两个 $T(n/2)$取代后得 $T(1)=1$; $T(2)=2$; $T(n)=2T(n/2)+4$，如表2.6所示。解开此数学式子即可知 $T(n)$。如何知道此数学式子的解呢？最简单的方法是用“猜”。

表2.6 数学式子 T(1)=1; T(2)=2; T(n)= 2T(n/2)+4 的值

n	1	2	4	8	16	32	64
$T(n)$	1	2	8	20	44	92	188

直观上看起来，$T(n)$都小于 n 的常数倍数，故 $T(n)$可能为 $O(n)$。若以 O 的角度分类，则这两个算法可归类于 $O(n)$这个时间复杂度等级的算法。

2.4　二维极点问题

若将2.2节寻找最大值的问题扩展到坐标平面上，则成为了二维极点问题（the 2-dimensional maxima finding）。

一个平面上的点的坐标可用两个整数(x, y)表示。当$x_1>x_2$且$y_1>y_2$时，我们说一个平面上的点(x_1, y_1)支配（dominate）另一个点(x_2, y_2)，即图2.2中右上方的点支配左下方的点。

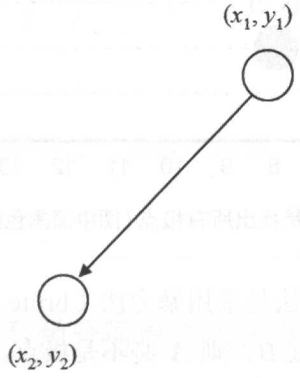


图 2.2　平面点(x1, y1)支配另一个平面点(x2, y2)

二维极点问题就是在一些平面上的点中找到那些没有被其他点支配的点，也就是极点（maxima）。直觉上，极点就是所有坐标点中那些位居右上方的一些坐标点，如表2.7和图2.3所示。

表 2.7　二维极点问题

问题	在平面上的 n 个点中，找到那些没有被其他点支配的点
输入	平面上的 n 个点 (2, 4)、(3, 10)、(5, 3)、(6, 8)、(8, 2) (10, 6)、(13, 5)、(15, 7)
输出	找到所有没有被其他点支配的点（极点） (3, 10)、(6, 8)、(15, 7)

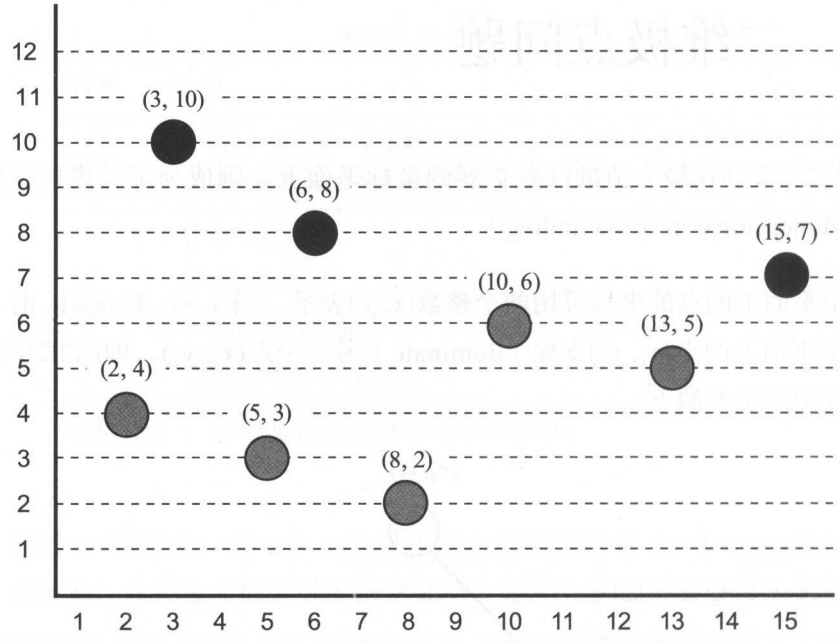


图 2.3　二维极点问题是找出所有极点（图中深黑色的点）

解决二维极点问题最简单的方法是采用暴力法（brute force method）。想法很简单，若某点 A 右上方有一点 B，则 A 必不是极点。因为极点的右上方一定没有其他点，如图2.4所示。

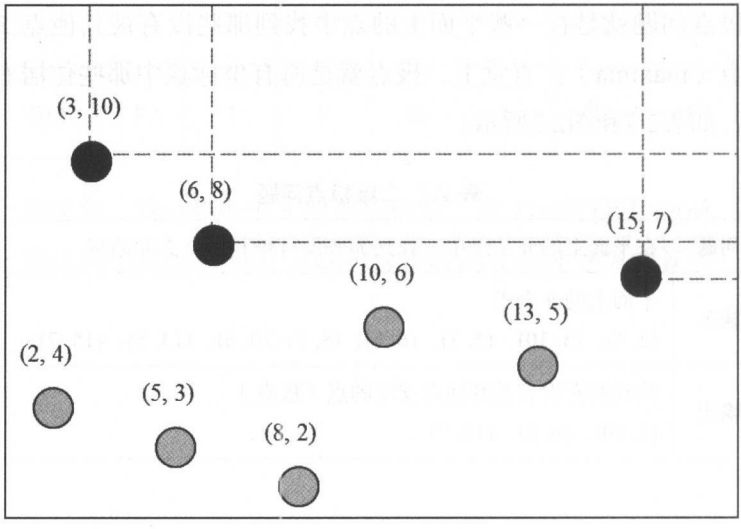


图 2.4　极点（深黑的点）右上方必没有其他点

此暴力法就是每一点 A 都和其他点 B 比较。若 B 在 A 的右上方，则排除 A 为极点的可能，最后留下（未被排除）的点即为极点。例如，在图 2.4 中，比较 (8, 2)和(10, 6)两点后，舍弃(8, 2)。(10, 6)并非极点，因为(10, 6)在和(15, 7)比较后，也被舍弃。此方法的时间复杂度取决于两点比较的总次数。

若给定 n 个点，任意两点需要比较一次，总共要比几次？

从 n 个点中任意选取两点的所有组合，也就是 $C(n, 2)$ 或 $\binom{n}{2}=\frac{n\times(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}=O(n^2)$。

这个暴力法虽然解决了二维极点问题，但是我们仍想知道是否有其他更快的方法。

这个问题可以利用分而治之法解决。

但是什么是分而治之法？

将一个问题分割成一些小问题并且递归地解决后，再利用这些小问题的解合并成原来问题的解。你认为设计分而治之法的第一步应该思考什么？

如何将二维极点问题进行分割。

怎样分割呢？最简单的分割方法是什么？

从中间分割成两半吧，如图2.5所示。

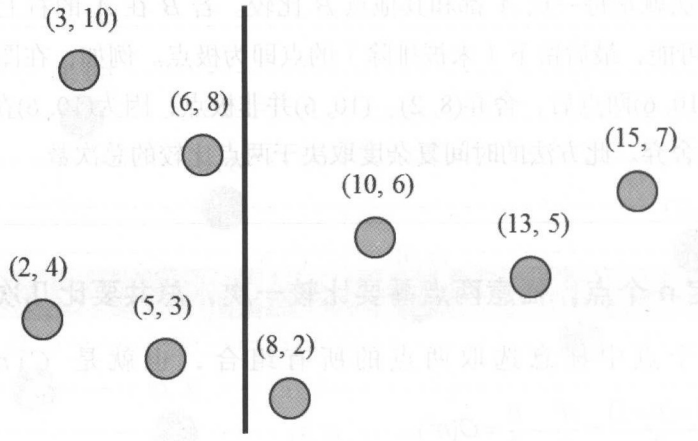


图 2.5 将平面的点分割成左右均等的两半

将平面的点分割成左右均等的两半后，分别（递归地）找到左边点的极点集合和右边点的极点集合，再将两组的极点合并成最后的解，如图2.6所示。

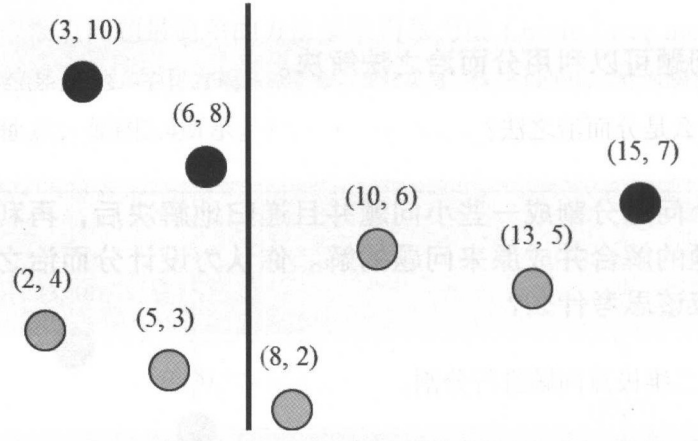


图 2.6 先找到左半边的极点和右半边的极点

在图 2.6 中欲合并左右两边的极点，只需要将两个集合执行并集操作就可以了。再多试几个例子，将会发现将左半边的极点和右半边的极点直接并集后，不一定就是最后的解，如图2.7所示。原因是在并集中出现了有问题的点（图2.7 中打 × 的点）。

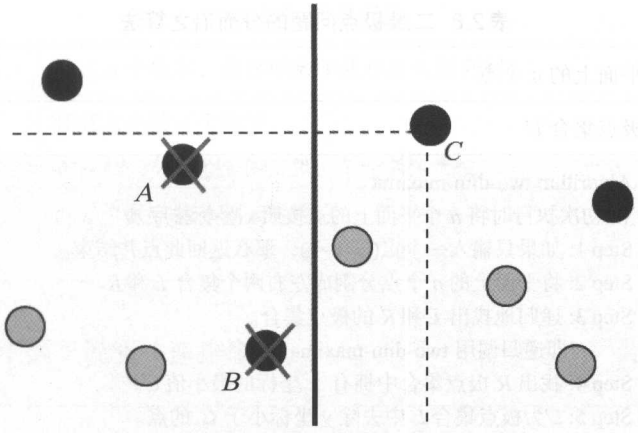


图 2.7　当左半边的极点被右半边的极点所支配时，必不是极点

有趣的是，有问题的点都在左半边，原因是有一些在左半边找到的极点有可能被右半边的极点所支配（图 2.7 中 A 和 B 被 C 所支配）。注意右半边点的 x 坐标必大于左半边点的 x 坐标，故当右半边点的 y 坐标也大于左半边点的 y 坐标时，右半边的点便支配左半边的点，这些左半边的点必不是极点。

因此在右边的极点集合中，最高（或最左）的极点为 C。当合并时，左边极点集合中高度小于 C 的点必不是极点（因为这些点必为 C 所支配)。如图 2.7 中的 A、B 就必须被除去，因为这两点被 C 所支配。最后合并的解答如图2.8 所示。此分而治之的算法如表 2.8 所示。

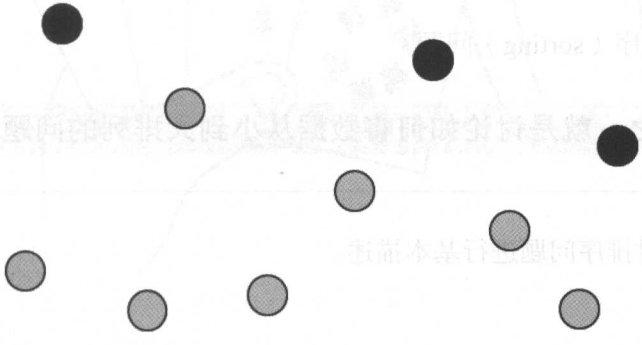


图 2.8　合并后的极点

表2.8 二维极点问题的分而治之算法

输入	平面上的 n 个点
输出	极点集合 U
步骤	Algorithm two-dim-maxima {/*初次执行时将 n 个平面上的点按照 x 坐标排序 */ Step 1: 如果只输入一个点(即 n=1)，那么返回此点并结束。 Step 2: 将平面上的 n 个点分割成左右两个集合 L 和 R。 Step 3: 递归地找出 L 和 R 的极点集合。 /*即递归调用 two-dim-maxima 两次*/ Step 4: 找出 R 极点集合中拥有 x 坐标的最小值 C。 Step 5: L'为极点集合 L 中去除 y 坐标小于 C 的点。 Step 6: 合并L'和 R 成为极点集合 U。 }

二维极点问题分而治之算法的时间复杂度是多少？若 $T(n)$为当输入 n 点时所需的运行时间，则 $T(n)$大约为 $2T(n/2)+O(n)$。若解开此递归式，则可得 $T(n)=O(n \log n)$。

2.5 快速排序法

快速排序法（Quicksort）是另一种典型的分而治之算法。

什么是排序（sorting）问题？

简而言之，就是讨论如何将数据从小到大排列的问题。

表2.9将对排序问题进行基本描述。

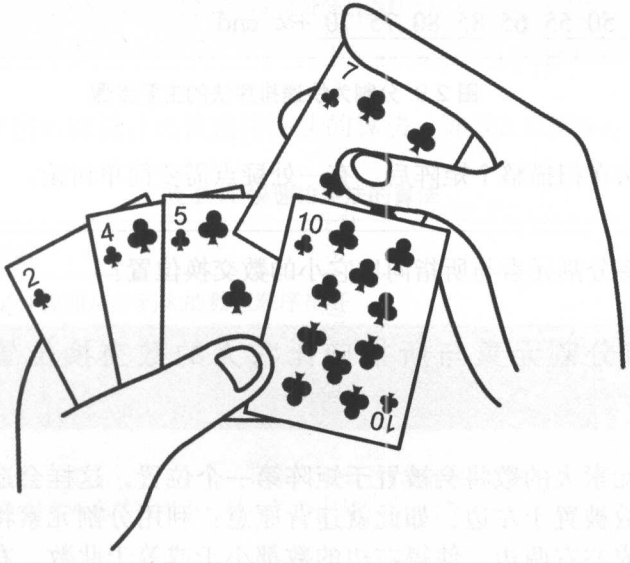
表 2.9　排序问题

问题	输入 n 个数字，将这些数字从小到大排列好
输入	任意大小的 n 个数字 65、70、75、80、85、60、55、50、45
输出	此 n 个数字从小到大的顺序 45、50、55、60、65、70、75、80、85

下面举一个例子说明快速排序法。

快速排序法的基本技巧是什么？

分割（partition）：用一个数将所有其他的数值分成左右两边，使得左边的数都小于或等于此数，右边的数都大于或等于此数。



下面的例子中，先将9个整数存储在数组中。取出矩阵中第一个数，当作分割元素（partitioning element）（即图2.9 中的 65）。从矩阵中第二个数开始向右寻找大于等于65 的数，同时从矩阵最后一个数开始向左寻找小于等于65的数。交换这两个数的位置，重复前述动作，直到左右两方的寻找箭头交叉，此时将分割元素，与所指向比它小的数交换位置。注意，图2.9 中数字下的箭头代表当前寻找的方向和位置。

```
(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)  i  p
65  70  75  80  85  60  55  50  45  +∞    2  9     /* 65 为分割元素*/
    →                           ←
65  45  75  80  85  60  55  50  70  +∞    3  8     /*70 与 45 交换后*/
        →                   ←
65  45  50  80  85  60  55  75  70  +∞    4  7     /*75 与 50 交换后*/
            →           ←
65  45  50  55  85  60  80  75  70  +∞    5  6     /*80 与 55 交换后*/
                →   ←
65  45  50  55  60  85  80  75  70  +∞    6  5     /*85 与 60 交换后*/
                ←   →
65  45  50  55  60  85  80  75  70  +∞    end      /*小数 60 与 65 交换*/
60  45  50  55  65  85  80  75  70  +∞    end
```

图 2.9 分割为快速排序法的主要步骤

分割算法在扫描整个矩阵后，有一处疑点需要简单讨论。

为何要将分割元素与所指向比它小的数交换位置?

如果将分割元素与所指向比它大的数交换位置，会怎样呢?

比分割元素大的数将会被置于矩阵第一个位置，这样会造成比分割元素大的数被置于左边，如此就违背原意：利用分割元素将所有其他的数值分成左右两边，使得左边的数都小于或等于此数，右边的数都大于或等于此数。”

表2.10 所示为此分割算法。

表 2.10 分割算法

输入	$a[m], a[m+1], \cdots, a[p-1]$。其中分割元素 $t = a[m]$
输出	将分割元素 t 置于 a 矩阵的适当位置 $a[q]$，使得 $a[m]$到 $a[q-1]$的值都小于或等于 t，$a[q+1]$到 $a[p-1]$的值都大于或等于 t

步骤	Algorithm partition(a, m, p)
	{
	Step 1: v:=a[m]; i:=m; j:=p;
	/* v 为分割元素，i 为矩阵开始的下标，而 j 为矩阵结束的下标*/
	Step 2: 重复以下步骤，直到结束。
	{
	repeat
	i:=i+1;
	until (a[i] >=v);　/*找到比分割元素大的数*/
	repeat
	j:=j-1;
	until (a[j] <=v);　/*找到比分割元素小的数*/
	if (i<j) then 交换 (a[i], a[j]);
	}
	until (i>=j);　/*以上两方寻找的下标交错*/
	a[m]:=a[j]; a[j]:=v; return j;
	/*将分割元素置于 a[j]处并且返回j*/
	}

利用分割算法可以设计出快速排序法的算法，如表2.11所示。

表 2.11 快速排序法的算法

输入	$a[p], \cdots, a[q]$
输出	$a[p], \cdots, a[q]$ 按照从小到大的数值顺序排好

步骤	Algorithm quicksort (p, q)
	{
	if (p<q) then
	{
	j:=partition(a, p, q+1);　/*执行分割算法(见表 2.10)*/
	/*分割元素所在的数组下标被返回并存储于变量j中*/
	/*以下两个子程序的调用就是以递归方式排序两个小矩阵的数值*/
	quicksort(p, j-1);
	quicksort(j+1, q);
	}
	}

2.6　快速排序法的时间复杂度

快速排序法的时间复杂度是多少？

当输入 n 个数字时，快速排序法所需的运行时间为 $T(n)$。利用分割元素会将整个矩阵（n 个数据）分割为两个未排序的小矩阵，之后再递归地调用两次快速排序法，来处理剩下的排序工作。

根据上述说明，可得 $T(n)=T(j)+T(n-j-1)+n+1$。此处，$T(j)$和 $T(n-j-1)$为两个小矩阵，是被递归调用两次快速排序法所需的时间，而 $n+1$ 是分割算法扫描矩阵（直到箭头交错）所需的比较（comparison）时间（见图2.9）。为了简化这个分析，在此我们仅考虑所需的比较（comparison）时间，并不包含交换（exchange）所需的时间。

若能解开此递归式，则可得快速排序法的时间复杂度 $T(n)$。显然，$T(n)$的值可能会因 j 值的不同而有所变化。下面我们将讨论快速排序法的最佳、最差及平均时间复杂度。

2.6.1　快速排序法的最佳时间复杂度

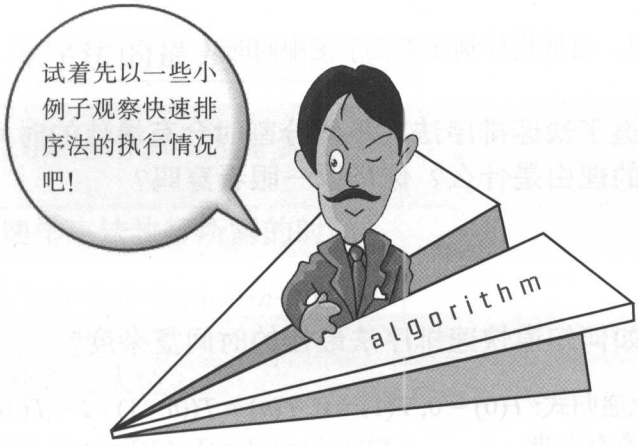
在哪种情况下，快速排序法执行得最快？

先想想什么是快速排序法执行所需的时间。

即递归式 $T(n)=T(j)+T(n-j-1)+n+1$ 的解。

当 j 为何值时，$T(n)$最小？

可能是 $j=1$ 或 n，或 $j=\lfloor n/2 \rfloor$或$\lceil n/2 \rceil$。



显然，上述讨论的答案取决于以下两式的解:

(1) $T(0)=0, T(1)=0, T(n)=T((n-1)/2)+T((n-1)/2)+n+1$

(2) $t(0)=0, t(1)=0, t(n)=t(1)+t(n-2)+n+1$

试着将这两式 $T(n)$和 $t(n)$前几项的值列出，如表 2.12所示。观察此表可知，这两式的值随 n 的变大而变大（递增性质）。另一个现象是，当 n 变得足够大时($n\geq 6$)，$t(n)$的值总是大于 $T(n)$的值。

表 2.12 递归式 $T(n)$和 $t(n)$ 前几项的值

n	0	1	2	3	4	5	6	7	8	9	10	11	12
$T(n)$	0	0	3	4	8	12	14	16	21	26	31	36	39
$t(n)$	0	0	3	4	8	10	15	18	24	28	35	40	48

最小值可能是 $T(n)$，最大值可能是 $t(n)$。

这表示在何种情况下，快速排序法执行得最快?

在 j 大约是 $n/2$ 时，会有最小值的 $T(n)$值，也就是快速排序法执行最快的时候。

这表示如何分割，快速排序法才会执行得最快?

平均分割，也就是分割元素位于正中间时。

我们知道了快速排序法在平均分割时会有最佳的时间复杂度，但真正的理由是什么？你可以一眼看穿吗？

嗯……

或者，如何知道快速排序法最佳的时间复杂度？

可解开此递归式：$T(0) = 0, T(1) = 0, T(n) = T((n - 1) / 2 + T((n - 1) / 2) + n + 1$。但好像有点难。

你可以想象一个类似的更容易的问题吗？

可以。递归式：$T(0) = 0, T(1) = 0, T(n) = T(n /2) + T(n /2) + n + 1 = 2T(n / 2) + n + 1$ 比较简单，尤其是当 n =2k 时。

解开此简单递归式的一个方法是利用重复展开。

快速排序法最佳时间复杂度的近似解如下：

$T(0)=0, T(1)=0,$
$T(n)=2T(n/2)+n+1$, 当 $n=2^k$。
$=2(2T(n/4)+n/2+1)+n+1$ (因为 $T(n/2)= 2T(n/4)+n/2+1$)
$=4T(n/4)+n+2+n+1$
$=4(2T(n/8)+n/4+1)+n+2+n+1$
$=8T(n/8)+n+4+n+2+n+1$
$=16T(n/16)+n+8+n+4+n+2+n+1$
.
.
.
$=2^k \times 0+\log_2 n \times n+(2^{k-1}+ \ldots +4+2+1)$
$=\log_2 n \times n+(2^{k-1}+ \ldots +4+2+1)$
$=\log_2 n \times n+n-1$
$=O(n \log n)$

注意　$T (0) = 0, T(1) = 0, T (n) = T ((n - 1)/ 2 + T ((n - 1)/ 2) + n + 1$ 才是快速排序法的真正最佳时间复杂度。然而其解可利用数学归纳法获得，而且也正好是 $O (n \log_2 n)$。

2.6.2　快速排序法的最差时间复杂度

快速排序法的最差时间复杂度是多少？

什么是快速排序法执行所需的时间？

即此递归式 $T(n)=T(j)+T(n-j-1)+n+1$ 的解。

何时 $T(n)$会有最大值？

应该是递归式 $T(n)=T(0)+T(n-1)+n+1$ 的解。

同样地，迭代此递归式即可得到快速排序法的最差时间复杂度 $O(n^2)$，过程如下：

$T(0)=0, T(1)=0,$
$T(n)=T(0)+T(n-1)+n+1$
$=0+T(n-1)+n+1$
$=T(n-1)+n+1$
$=(T(n-2)+n)+n+1$, 因为 $T(n-1)=T(n-2)+n$
$=T(n-2)+n+n+1$
$=(T(n-3)+n-1)+n+n+1$
$=T(n-3)+(n-1)+(n)+(n+1)$
.
.
.
$=T(1)+3+4+\cdots+(n-1)+(n)+(n+1)$
$=0+(3+n+1)(n-1)/2$
$=(n+4)(n-1)/2$
$=n^2/2+3n/2-2$
$=O(n^2)$

2.6.3　快速排序法的平均时间复杂度

最坏和最好的情况不会常常出现，那么一般情况下，快速排序法的表现如何？快速排序法的平均时间复杂度又是多少？

根据前两小节的讨论，快速排序法的执行速度和分割元素所在的位置有密切的关系。简单地说，当分割最平均时，快速排序法执行得最快；相反，当分割最不平均时，执行得最慢。当输入的数值无规律时，这两种情况应该不会每次都发生。倘若每一种分割出现的概率都是相同的（如图2.10 到图2.14 中 ↓ 所示的位置），那么快速排序法所需平均（期望）执行的时间是多少？

首先，矩阵共存储 n 个数值，当分割元素位于矩阵第一位时，其执行时间为 $T(n)=T(0)+T(n-1)+n+1$。

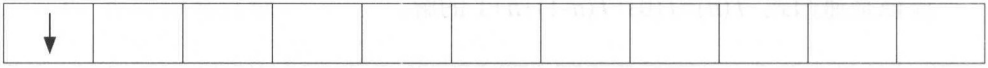


图 2.10 分割元素落在矩阵的最左边位置

当分割元素位于矩阵第二位时，其运行时间为 $T(n)=T(1)+T(n-2)+n+1$。

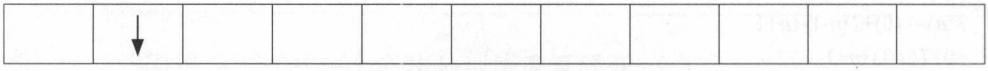


图 2.11 分割元素落在矩阵左边第二个位置

当分割元素位于矩阵第三位时，其运行时间为 $T(n)=T(2)+T(n-3)+n+1$。

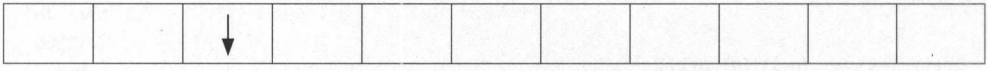


图 2.12 分割元素落在矩阵左边第三个位置

以此类推，当分割元素位于矩阵最后第二位时，其运行时间为 $T(n)=T(n-2)+T(1)+n+1$。

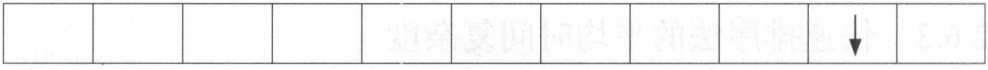


图 2.13 分割元素落在矩阵右边第二个位置

当分割元素位于矩阵倒数第一位时，其运行时间为 $T(n)=T(n-1)+T(0)+n+1$。

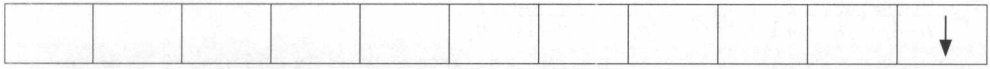


图 2.14 分割元素落在矩阵最右边的位置

假设每一种情况发生的概率都是一样的（即1/n），那么快速排序法的平均时间复杂度可将每一种情况所需的运行时间加总后再除以 n，得到以下递归式：

$$T(n)=((T(0)+T(n-1)+n+1)+(T(1)+T(n-2)+n+1)+\cdots+(T(n-2)+T(1)+n+1)+(T(n-1)+T(0)+n+1))/n$$

稍作整理可得：

$$T(n)=n+1+2/n\times(T(0)+T(1)+\ldots+T(n-1))$$

解开此递归式后，可得 $T(n)=O(n\log n)$。

快速排序法的平均时间复杂度详细分析如下：

$T(0)=T(1)=0$

$T(n)=n+1+2/n\times(T(0)+T(1)+\ldots+T(n-1))$

将上式等号的左右各乘上 n 可得：

$nT(n)=n(n+1)+2(T(0)+T(1)+\ldots+T(n-1))$ ……(A)

将上式所有的 n 用 $n-1$ 代入，得出另一等式：

$(n-1)T(n-1)=(n-1)n+2(T(0)+T(1)+\ldots+T(n-2))$ ……(B)

将(A)-(B)，可得一个较精简的递归式：

$nT(n)-(n-1)T(n-1)=n(n+1)-n(n-1)+2(T(0)+T(1)+\ldots+T(n-1)-(T(0)+T(1)+\ldots+T(n-2)))$

$=2n+2T(n-1)$

整理一下可得：

$nT(n)=(n+1)T(n-1)+2n$

再将上面的等式左右各除以 $n(n+1)$ 可得：

$$\frac{T(n)}{(n+1)}=\frac{T(n-1)}{n}+\frac{2}{n+1}$$

迭代此递归式可得：

$$\frac{T(n)}{(n+1)}=\frac{T(n-1)}{n}+\frac{2}{n+1}$$

$$=\frac{T(n-2)}{n-1}+\frac{2}{n}+\frac{2}{n+1}\text{，因为}\frac{T(n-1)}{(n)}=\frac{T(n-2)}{n-1}+\frac{2}{n}$$

$$=\frac{T(n-3)}{n-2}+\frac{2}{n-1}+\frac{2}{n}+\frac{2}{n+1}\text{，因为}\frac{T(n-2)}{(n-1)}=\frac{T(n-3)}{n-2}+\frac{2}{n-1}$$

$$=\frac{T(1)}{2}+\frac{2}{3}+\frac{2}{4}+\ldots+\frac{2}{n+1}=\frac{T(1)}{2}+2\sum_{3\leqslant k\leqslant n+}\frac{1}{k}=2\sum_{3\leqslant k\leqslant n+1}\frac{1}{k}$$

换言之，$T(n)=(n+1)\times 2\sum_{3\leqslant k\leqslant n+1}\frac{1}{k}$

而 $\sum_{3\leqslant k\leqslant n+1}\frac{1}{k}\leqslant\int_2^{n+1}\frac{1}{x}dx=\log_e(n+1)-\log_e 2$

因此最后可得：$T(n)\leqslant 2\times(n+1)\times[\log_e(n+1)-\log_e 2]=O(n\log n)$

2.7　寻找第 k 小值问题

最后一个例子是，在 n 个数中寻找第 k 小值问题，如表 2.13所示。当 k=1 或 n 时，就是找最小值或最大值问题，在第2.2节中，我们利用一个循环便可轻易解决。但当 k 不固定时，如何有效地解决此问题呢?

表 2.13　寻找第 k 小值问题

问题	输入 n 个数，请输出第 k 小值
输入	n 个未排序好的数和 k n =8, 72, 34, 12, 84, 21, 45, 58, 63 k=4
输出	第 k 小值 34 为第 4 小值

如何在 n 个数中寻找第 k 小值?

可先将 n 个数从小到大排序（sort）后，再从矩阵第 k 个位置获取。

如此所需的时间复杂度为多少?

若使用一般的排序法，则需要 $O(n \log n)$。

有可能更快吗? 比如 $O(n)$。如何在 $O(n)$时间内，寻找第 k 小值?

下面进行介绍。

2.7.1　寻找第 k 小值的 $O(n)$ 算法

本节的重点是设计一个可以找到任意第 k 小值的 $O(n)$算法。根据上述讨论，显然不可以利用排序法（否则所需的时间将达到 $O(n \log n)$)。

此算法需要的技巧源自于快速排序法中的分割算法（参考第 2.5节）。首先，利用快速排序法中的分割算法将整个矩阵分成“比分割元素大”和“比分割元素小”的两个集合。然后，如图2.15所示，分割元素 65 是第 5 小的数，若 k=5，则此数为解；若 k>5，则需在右边集合中寻找，此时左边集合可略过不必找；若 k<5，则需在左边集合中寻找，此时右边集合可略过不必找。

递归地使用以上步骤，即可找到第 k 小值。问题是，这样的做法所需的最长时间等同于快速排序法在最差情况下的时间复杂度 $O(n^2)$。

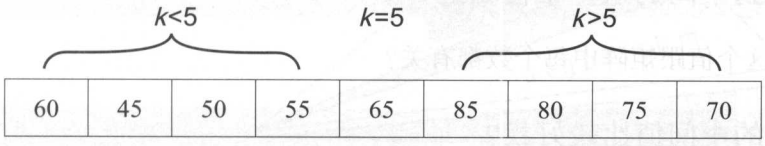


图 2.15 分割元素的位置决定寻找第 k 小值的范围

上述方法的缺点是什么?

应该还是使用分而治之法时，分割元素并未平均分割整个矩阵。

未平均分割的原因何在?

因为分割元素有时不是太大就是太小。

最好的分割元素是什么?

找中间的值当作分割元素。

如何找中间的值?

可先将 n 个数从小到大排序（sorting）后，再从矩阵中间位置获取。

这样的算法可以在 $O(n)$内完成吗?

糟了，不行！排序就花掉了 $O(n \log n)$ 的时间。

有其他方法可以较快地找到中间的值吗？如果你一时不能解决此问题，可先尝试一些相关的问题。可以想象到一个更容易的相关问题吗?

找到接近中间的值不知道有没有帮助?

如何很快地找到接近中间的值?

好像很难?

要找到中间的值，难在哪里?

好像这个值跟矩阵中每个数都有关?

怎样的中间值比较好找?

如果从 5 个数中找中间值就容易多了。

这样可以帮助你找到中间的值或较接近中间的值吗?

n 个数太多了，若是只有5个数就好找了。也许将 n 个数分成每5个数一组……

然后呢?

将每一组的中间值找到。

共有几组中间值? 然后呢?

大约 $n/5$ 组，然后从这些值中找到其中间值。

这样的值拿来当作分割元素，不知道效果好不好?

下面进行介绍。

以下算法就是利用此想法快速获取大致接近中间的值。例如，将n 个数排列成每 5 个数一行，若最后一行不足，则自成一行，如图2.16所示。

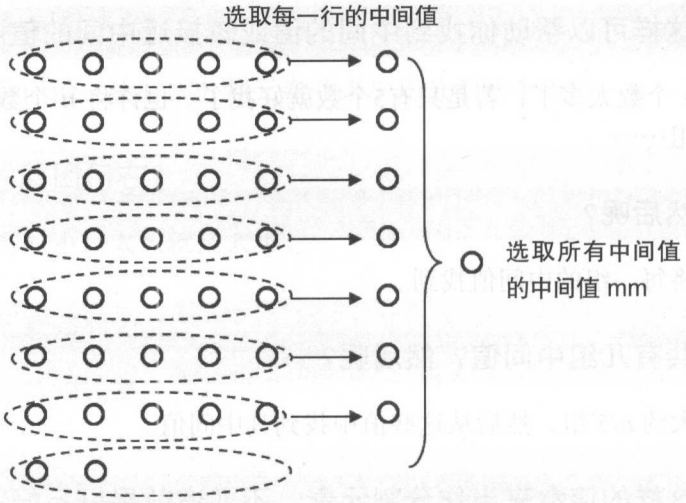


图2.16 将 n 个数排列成每 5 个数一行后，取其中间值的中间值

每行 5 个数，分别找出其中间值（共有 $\left\lfloor \frac{n}{5} \right\rfloor$ 个数）后，再从这些中间值中取出其中间值 mm。表2.14是利用 mm 当作分割元素，寻找第k 小值的算法。

表 2.14 寻找第 k 小值的 $O(n)$算法

输入	矩阵 $a[low:up]$
输出	$a[low:up]$中第 k 小值

步骤	Algorithm selection (a, k, low, up)
	{
	Step 1: 计算矩阵中数值的个数 n:=up-low+1;
	Step 2: if (n<=5) then 排序 a[low:up]并返回第 k 小值;
	Step 3: 将 a[low:up]分割成 [n/5] 行;每行数目为 5(最后一行可能不足5);
	Step 4: 将每行的中间值找出，并存储于矩阵 m[i](1≤i≤[n/5]) ;
	Step 5: 递归调用selection寻找矩阵 m[i]中的中间值;即从 [n/5] 个 m[i]中找出第[[n/5]/2]小值，并存入mm中(即执行 mm:=Selection(m, [[n/5]/2], 1, [n/5]));
	Step 6: 利用 mm 当作分割元素，执行分割算法 Partition(a, low, up)后，返回mm存于矩阵a的第j个位置(即下标为 j);
	Step 7: if (k=(j-low+1)) then 返回第k小值为a[j];
	else if (k<(j-low+1)) then 第k小值为递归调用
	selection(a, k, low, j-1)后返回的值;
	else 第k小值为递归调用 selection(a, k-(j-low+1), j+1, up)后返回的值;
	}

注意　从中间值中取出中间值 mm 的步骤（即 Step 5）是递归地调用寻找第 k 小值的算法，只是此时的 k 值是中间值总数的一半。

2.7.2　寻找第 k 小值算法的时间复杂度分析

寻找第 k 小值的 $O(n)$算法的关键是，所选的分割元素 mm 必须足够靠近整个矩阵的中间值。如此在分割后，无论向右找（比 mm 大的值）或向左找（比 mm 小的值），都可保证至少有足够多（至少大约 $n/4$）的数不必寻找，如图2.17所示。

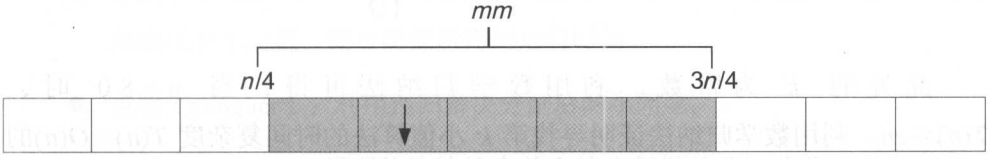


图2.17 分割元素 mm 为介于矩阵 a 第 n /4 到第 3n/4 小的数

下面说明分割元素mm 属于矩阵 a 中“大约介于第 $n/4$ 到第 $3n/4$ 小的数”的理由。因为 mm 是 $m[i]$ ([n/5] 列)中的中间值，所以 $m[i]$ 中必有 [[n/5] / 2]的值大于或等于 mm，如图2.18 中间的椭圆形所示。除了 mm 这行和可能未排满的最后一行，在剩下的 ([[n/5] / 2] – 2) 行中，每行有 [5/2] = 3 的值大于或等于 $m[i]$，如图2.18 右下方的长方形所示。如此可知，至少有 3 ×([[n/5] / 2] - 2 ≥ $\frac{3n}{10}$ - 6个数大于或等于 mm。

同理，也有多于 $\frac{3n}{10}-6$ 的值小于或等于 mm，如图2.18左上方的长方形所示。因此在 Step 7（见表2.12）递归调用 selection 时的输入矩阵 a 中，最多有 $\frac{7n}{10}+6$(即 $n-(\frac{3n}{10}-6)$) (接近 $3n/4$)个数字。

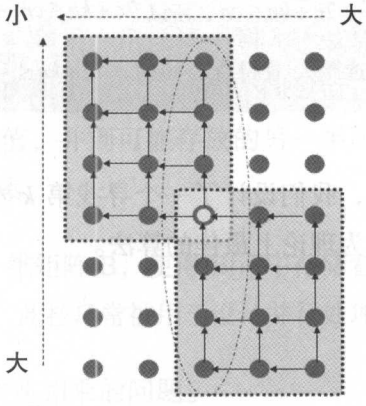


图 2.18 至少有 $\frac{3n}{10}-6$ 个数大于或等于 mm(右下方的长方块)，同时至少有 $\frac{3n}{10}-6$ 个数小于或等于 mm(左上方的长方块)

注意　**图 2.18 的数值排列是假设左上方的值都小于右下方的值。在算法中，我们并不需要真正进行这样的排列，毕竟此图只是用来说明的，分割元素 mm 的确足够靠近整个矩阵的中间值。**

表2.12中的步骤，除了 Step 5 和 Step 7 外，都可在 $O(n)$内完成。Step 5 需要 $T([\ n/5\ ])$的时间，而 Step 7 需要 $T(\frac{7n}{10}+6)$的时间。因此寻找第 k 小值算法的时间复杂度 $T(n)$可用以下式子表示：

$$T(n)\leqslant T(\lceil n/5\rceil)+T(\frac{7n}{10}+6)+kn$$

此处的 k 为常数。利用数学归纳法可得，当 n＞80 时，$T(n)\leqslant cn$。利用数学归纳法证明寻找第 k 小值算法的时间复杂度 $T(n)= O(n)$的过程如下：

定理 2.1 $T(n)\leq T(\lceil n/5\rceil)+T(\frac{7n}{10}+6)+kn$，因此 $T(n)<=cn$ 当 $n>80$

证明：假设小于 n 时，本定理成立，那么：

$$T(n)\leq T(\lceil n/5\rceil)+T(\frac{7n}{10}+6)+kn\leq c(n/5+1)+c(\frac{7n}{10}+6)+kn$$

$$\leq\frac{2cn}{10}+\frac{7cn}{10}+7c+kn\leq\frac{9cn}{10}+7c+kn$$

$$\leq cn$$

因为 $\frac{9cn}{10}+7c+kn\leq cn$，所以 $7c+kn\leq cn-\frac{9cn}{10}=\frac{cn}{10}$

因此我们选择 c，使得当 $n>80$ 时，$7c+kn\leq\frac{cn}{10}$ 为真

因此，我们设计了一个寻找第 k 小值问题的线性（即 $O(n)$)时间复杂度的算法，此为理论上最佳的算法。

2.8　分而治之法的技巧

什么问题可利用分而治之法来解决呢?

这个问题显然不容易回答。不过，一旦一个问题可被分而治之法解决，也常隐含此问题可被并行处理。最后，提醒使用分而治之法时的注意事项。

- 分割（divide）：尽可能快速地将问题平均地分割。若能将问题平均地分割成几个小问题，则可降低所需的运行时间。
- 克服（conquer）：利用递归解决分割后的小问题。
- 合并（merge）：尽可能快速地合并上述小问题的解，成为原问题的解。合并时所需的时间会影响整个算法所需的运行时间。

还有什么问题可被分而治之法解决?

下面列出一些可被分而治之法解决的常见问题，供读者参考。

（1）合并排序（merge sort）：是一个利用矩阵中邻近的数字两两合并，将数字从小到大排列的方法。

（2）二分搜索法（binary search）：利用一组已从小到大（或从大到小）排序好的数值进行特定数值的搜索。每次取出搜索区间的中间数值，与欲搜索的值进行比较后，舍弃不可能的一半数值，并到可能存放的另一半区域继续寻找。直到寻得或确定不存在，才停止搜索。

（3）前缀和（prefix sum）：计算一维矩阵 B，使得 B[i] 中存储着从 A 矩阵中第一个值加到第 i($1 \leqslant i \leqslant n$)个值的总和。此技巧常被用于设计并行处理的算法。

（4）矩阵相乘问题：将两个矩阵快速相乘的问题。

（5）邻近配对问题（closest pair problem）：找出平面点中最靠近的两点的距离。

（6）凸包问题（convex hull problem）：找出一个包含所有输入平面点的最小凸多边形的问题。

（7）维诺图（Voronoi diagrams）：将每一个平面点 x 分割落于单独的一个区域，使得另一个新加的平面点 y 落于此区域时，其最近的点为 x。此算法技巧可用于无线移动设备 (y) 快速地寻找最近的基站 (x)。

学习效果评测

1. 设计一个分而治之的程序解决二维极点问题。

输入：

```
8                    (平面点的个数)
2 4                  (以下是每个点的x和y坐标)
3 10
5 3
6 8
8 2
10 6
13 5
15 7
```

输出：

```
3  10
6  8
15 7
```

2. 根据 O 的定义，证明当 $T(n)=T(n/2)+T(n/2)+1$时，$T(n)=O(n)$。

3. 设计一个程序执行快速排序法(quicksort)。

输入：

```
65 70 75 80 85 60 55 50 45
```

输出：

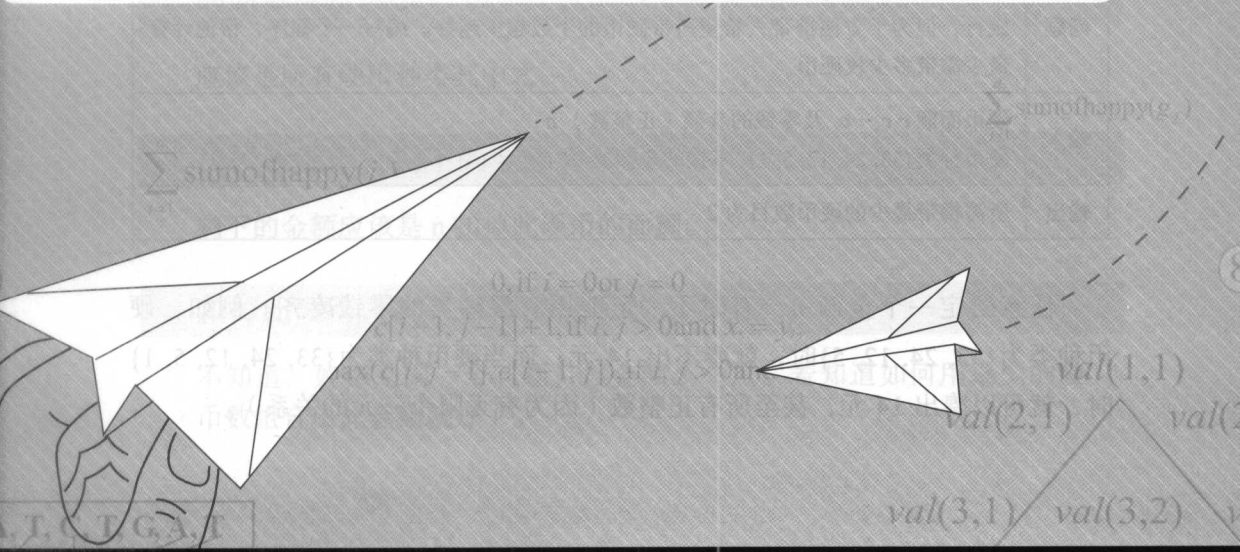
```
45 50 55 60 65 70 75 80 85
```

动态规划

章节大纲

“岱宗如何”是泰山派剑法的绝艺，要旨不在右手剑招，而在左手的算数。左手不住屈指计算，算的是敌人所处方位、武功门派、身形长短、兵刃大小以及日光所照高低等，计算极为繁复，一经算准，挺剑击出，无不中的。

金庸 笑傲江湖



3.1　何谓动态规划

什么是动态规划（dynamic programming）？

简而言之，就是计算并存储小问题的解，并将这些解组合成大问题的解。

当大问题的解可利用小问题的解组合计算得知时，若将可能用到的小问题的解先算出并存储起来，则可缩短计算大问题的解的时间。

乍看之下，动态规划还蛮暴力的。使用动态规划常经过烦琐的计算，最后一举解决问题，有一点像金庸的武侠小说《笑傲江湖》中的绝招“岱宗如何”。但是需注意的是，小问题的解一旦被计算出并存储后，就不会再被重复计算。当动态规划运用得当时，可高效率地解决问题。

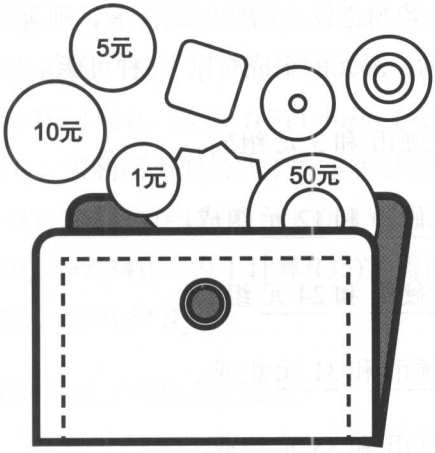
3.2　换零钱

第一个例子是换零钱问题，问题描述如表3.1所示。

表 3.1　换零钱问题

问题	老王是一位富翁，拥有无限硬币，但硬币种类有限。老王想带一定金额的零钱 n 去旅行，但为了方便携带，希望所带硬币的个数越少越好。编写一个程序，帮他计算最少需带多少枚硬币
输入	硬币面额 $c_1c_2\cdots c_k$ 及零钱的金额（正整数）n 硬币种类{33, 24, 12, 5, 1}, 零钱的金额 n=36
输出	所需携带最少的硬币数目为 2

任意给定一个金额 n，有时不一定能用有限种类的零钱凑齐。例如，硬币种类为{33, 24, 12, 5}时，就凑不出 14 元；而当硬币种类为{33, 24, 12, 5, 1}时，就可以凑出 14 元，甚至所有正整数（因为有无限个一元的关系）。



如果换零钱的策略为尽量先兑换大金额的零钱，那么会发现这个方法有时并不能换到最少的硬币数目。如表3.1的例子，就会换成4个硬币（36 = 33 + 1 + 1 + 1），而非最佳的两个（36 = 24 + 12）。

怎样才可以使用最少的硬币，组合出总和为 n 的金额呢?

所需最少的硬币数目是多少?

不知道。

假设我们知道最佳(少)的硬币组合，猜想其中的任意一个硬币的面额会是多少?

应该是所有硬币种类其中之一。

扣掉此硬币，剩下的金额为多少?

剩下的金额应该是 n 扣掉此硬币的面额。

扣掉此硬币，剩下所需的硬币数为多少?

不知道。好像是同一个问题，但金额小一点。若知道如何用最少的硬币数组合出此金额就好了。

最后一句话暗示：若知道较小问题的最优解，则有助于解决大问题。按表3.1中换零钱问题的范例，36元的组成有以下5种可能：

（1）由一个33元硬币和3元组成。

（2）由一个24元硬币和12元组成。

（3）由一个12元硬币和24元组成。

（4）由一个5元硬币和31元组成。

（5）由一个1元硬币和35元组成。

为什么是5种？

因为共有5种硬币。

在5种组成方法中，哪一种需要的硬币数最少？

不知道。因为不清楚 3、12、24、31、35 这些金额需要的最少硬币数。

可以事先计算出 3、12、24、31、35 分别需要多少硬币。

从以上5种可能的兑换方法中挑选所需硬币数最少的一种，即最佳换零钱的方式。但是需要事先计算出这5种金额的最少硬币数。

根据上述讨论，若让函数 $f(n)$ 代表组合金额为 n 所需要最少的硬币数，则下列等式是正确的。

```
f(36)=min{1+f(3), 1+f(12), 1+f(24), 1+f(31), 1+f(35)}
```

此处，*min* 是从一个集合中选取最小值 (minimum value) 之意。

动态规划的技巧是：为了计算 $f(36)$，需事先利用类似的方式计算 $f(3)$、$f(12)$、$f(24)$、$f(31)$和$f(35)$。下面以 $f(31)$ 和 $f(35)$为例进行说明。因为 31=24+7=12+19=5+26=1+30，所以 $f(31)=min\{1+f(7), 1+f(19), 1+f(26), 1+f(30)\}$。为了计算$f(31)$，需利用类似的式子，事先计算出$f(7)$、$f(19)$、$f(26)$和$f(30)$。同样，因为 35=33+2=24+11=12+23=5+30=1+34，所以$f(35)=min\{1+f(2), 1+f(11), 1+f(23), 1+f(30), 1+f(34)\}$。为了计算$f(35)$，需使用类似的式子，事先计算出$f(2)$、$f(11)$、$f(23)$、$f(30)$ 和 $f(34)$。

总而言之，下面的两个式子显然是正确的。

（1）$f(0)=0$。

（2）$f(n)=1+min\{f(n-c_1), f(n-c_2), \cdots, f(n-c_k)\}$。

当 $n>c_i(1<i<k)$且硬币面额为c_1 或 $c_2\cdots$或 c_k时，动态规划的技巧会将 $f(n-c_1)$, $f(n-c_2)$, $\cdots$, $f(n-c_k)$ 先行计算后存储起来，再按照上述式子计算出 $f(n)$ 的值（即找出 $f(n-c_1)$, $f(n-c_2)$, $\cdots$, $f(n-c_k)$ 中的最小值后，再加上 1）。表 3.2 将列出换零钱问题的算法。

表 3.2 换零钱问题的算法

输入	硬币面额种类 $c_1c_2\cdots c_k$ 及欲携带零钱的金额 n (正整数)
输出	组合金额总和为 n 的最少硬币数目

步骤	Algorithm coin_changing
	{
	Step 1: 当z为0或负数时，令 $f(z)=0$。
	Step 2: for k=1 to n
	{
	计算 $f(k)=1+min\{f(k-c_1), f(k-c_2), \cdots, f(k-c_k)\}$。
	}
	}

换零钱问题的算法在于利用一个循环和前页中间的式子(2)先计算比n 小的所有金额的解（即该金额所需的最少硬币数）。最后利用这些解再计算出金额 n 的解。

3.3 数字金字塔

第二个例子是数字金字塔，如表 3.3所示。

表 3.3 数字金字塔

问题	老王到埃及看金字塔，想从金字塔底向上走到顶。可是每一条路所花的时间都不一样长。请帮他找到登上金字塔最快的路（所费时间为路径上数字的总和）。注意每一个向上的路径只有左上和右上的两条路
输入	叠成金字塔的正整数 （注意金字塔的高度不固定） 45 20　33 34　18　30 14　45　09　11
输出	从底部任意一处走到金字塔最顶端的最快路径 45+20+18+9=92，路径如下： 45 20　33 34　18　30 14　45　09　11

这一题可用动态规划来解吗?

怎样判断一个问题可不可以用动态规划来解?

为何换零钱问题可以被成功地解决?

利用较小问题的最优解组成大问题的最优解。

这题如果要使用同样的技巧，下一步要做什么?

找出小问题的最优解和大问题的最优解之间的关系。

3.3.1　找出大问题和小问题之间的关系

弄清楚大问题和小问题之间的关系，显然是解决本题的关键。如图3.1 所示，抵达金字塔顶点45的路径只有两种可能：(A)经过 20；(B)经过 33。

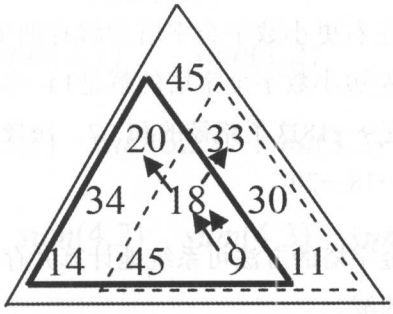


图 3.1　左右数字金字塔的解有助于找到大数字金字塔（以 45 为顶）的解

如果从底部走到 20 和从底部走到 33 的最快路径都事先被找到，只要从中选一条最快的路，直接登上顶端45就是最佳答案了。如果事先计算得知，经过 20 的最快路径 9→18→20 所费的时间为 47，而经过 33 的最快路径 9→18→33 所费的时间为 60，显然经过 20 的路径是两条中的较小者。也就是抵达金字塔顶点45最快的路径是一条先经过 20 的路径。

因此，抵达大金字塔顶点 45 最快的路是从左边小金字塔顶点 20 最快的路和右边小金字塔顶点 33 最快的路中选最短的。这就是大问题的最优解和小问题的最优解之间的关系。

如何知道小一些的数字金字塔的最优解？

知道更小一些的数字金字塔的最优解，将有助于解决这个问题。

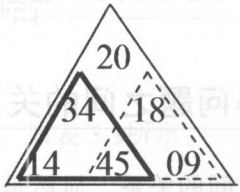


图 3.2 左右更小数字金字塔的解有助于找到数字金字塔（以 20 为顶）的解

如图 3.2 所示，左右更小数字金字塔的解有助于找到数字金字塔（以 20 为顶）的解。显然，左边小数字金字塔的解是14→34这条路径的和 48，而右边小数字金字塔的解是9→18这条路径的和 27。比较大小之后，抵达金字塔顶点 20 的最快路径是9→18→20。

因此，解决数字金字塔的方法可系统地计算并存储所有小问题的解，并将这些解组合成大问题的解。

3.3.2 数字金字塔问题的数据结构

下面讨论所需要的两个数据结构。

1. 矩阵变量 *val*(*x*, *y*) 存储数字金字塔第 *x* 层、第 *y* 个元素的值(value)。

例如，图3.3 中 val (3, 2) 存储了数字 18。

$$val(1,1)$$
$$val(2,1) \quad val(2,2)$$
$$val(3,1) \quad val(3,2) \quad val(3,3)$$
$$val(4,1) \quad val(4,2) \quad val(4,3) \quad val(4,4)$$

图 3.3 矩阵 val(x, y)存储数字金字塔第x 层、第 y 个元素的值

2. 矩阵变量 *sum*(*x*, *y*)存储从底层走到以第 x 层、第 *y* 个元素为金字塔顶端的最快路径值。

例如，在图 3.4 中， *sum*(3, 2)存储以18=*val*(3, 2)、45=*val*(4, 2)及 9=*val*(4, 3)三个数形成的金字塔（即图3.2中的右三角形）的最快路径的值（即 *sum*(3, 2)=18+9=27)。换句话说，*sum*(*x*, *y*)存储所有大小金字塔问题的最优解。

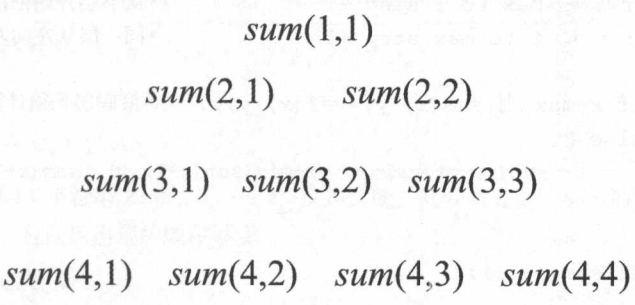


图3.4 矩阵 sum(x, y) 代表从底层走到以第 x 层、第 y 个元素为金字塔顶端的最快路径值

根据以上讨论，下列式子显然是正确的。这里，子程序min(*a*, *b*)返回*a*、*b*中的较小值。

```
sum(x, y)=val(x, y)+min(sum(x+1, y), sum(x+1, y+1)) if x 不是底层。
sum(x, y)=val(x, y) if x 是底层。
```

注意　上面这个式子准确地描述了大问题和小问题之间的关系。

最后，注意在计算 *sum*(*x*, *y*) 值时，可以从底层开始向上计算，即采取自下而上（bottom-up）的方式。理由很简单，因为上一层的值 *sum*(*x*, *y*)需要利用下一层的值 *sum*(*x*+1, *y*)和 *sum*(*x*+1, *y*+1)计算得出，所以下一层的值要先被计算并存储。如此，利用动态规划求解数字金字塔的算法就可被设计出来了，如表 3.4所示。

表 3.4　数字金字塔的算法

输入	叠成金字塔的正整数存储于 *val*(*x*, *y*)
输出	从底部任意一个数走到金字塔最顶端的最快路径(所费时间为路径上数字的总和)

步骤	/* *max* 代表金字塔的层数*/
	Algorithm pyramid
	{
	for *x* ← max to 1 step -1　　/*从下层开始依次计算*/
	for *y* ← 1 to max step +1　　/*同一层从左向右计算*/
	{
	if *x*=max 则{*sum*(*x*, *y*)=*val*(*x*, *y*);}　/*最底层无须计算*/
	else {
	sum(*x*, *y*)=*val*(*x*, *y*)+*min*(*sum*(*x*+1, *y*), *sum*(*x*+1, *y*+1));
	}
	}
	输出 *sum*(1, 1);
	}

3.4　最长相同子字符串

利用动态规划的算法技巧解决了前两个问题。但是这个算法技巧的主要精神是什么呢？下面这句话非常关键。

利用小问题的最优解组成大问题的最优解。

动态规划解决问题的3个步骤应该是：

（1）此问题的“大问题的最优解”可以利用“小问题的最优解”求取。

（2）利用一个数学式子将“大问题的最优解”和“小问题的最优解”之间的关系清楚地表达出来。

（3）先将“小问题的最优解”计算出后存储下来，再利用它们算出“大问题的最优解”。

接下来介绍一个可应用于生命科学上的有趣问题：最长相同子字符串（the longest common subsequence），如表3.5所示。并展现如何遵循这3个步骤设计出一个动态规划的算法。

表 3.5 最长相同子字符串

问题	老王找到失散多年的兄弟。为确定两人的血缘关系，他决定将两人的基因进行对比，请编写一个程序比较两组基因，并找到这两组基因（字符串）共同拥有的基因组（最长相同子字符串）。注意存在这两组基因的子字符串，其先后顺序需一致。
输入	两组字符串 $X=<x_1, x_2, \cdots, x_m>$ $Y=<y_1, y_2, \cdots, y_n>$ $X=$ <A, T, C, T, G, A, T> $Y=$ <T, G, C, A, T, A>
输出	最长相同子字符串 $Z=<z_1, z_2, \cdots, z_k>$及其长度。此处所有 $z_i(1\leqslant i\leqslant k)$需同时出现于 X 和Y 中，且前后出现的顺序不变 $Z=$ <T, C, T, A>，4

下面将遵循上述3个步骤设计算法。

3.4.1 步骤1：检查大问题的最优解是否可以利用小问题的最优解求取。

若用表 3.5 中的范例说明步骤1，则是问：

X= <A, T, C, T, G, A, T> 和 Y= <T, G, C, A, T, A>的最长相同子字符串可否通过小一些的最长相同子字符串获得?

答案可以从下面两个稍小问题的解中选择一个获取。

（1）X^*=<A, T, C, T, G, A> 和 Y = <T, G, C, A, T, A> 的最长相同子字符串（即<T, C, T, A>）。

（2）X=<A, T, C, T, G, A, T> 和 Y^*= <T, G, C, A, T> 的最长相同子字串（即<T, C, A, T>）。

X= <A, T, C, T, G, A, T> 和 Y= <T, G, C, A, T, A>的最长相同子字符串只有两种可能：（1）X 最后没有字母T；（2）Y 最后没有字母 A（见图 3.5）。理由是最长相同子字符串必须同时出现在两个字符串中。而此范例中的 X 和 Y 尾部的字母不同（即 T ≠ A），故 X 和 Y 的最长相同子字符串不可能同时拥有 X 和 Y 的尾部。

若事先计算出上述两个（(1)和(2)）最长相同子字符串的答案，则其中较长的相同子字符串为 X 和 Y 的最长相同子字符串。此大问题的最优解包含小问题的最优解的性质，称为最优子结构（optimal substructure）。

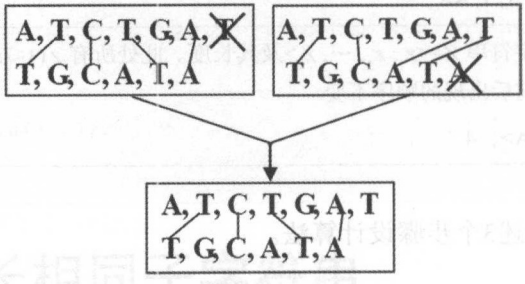


图3.5 最长相同子字符串问题的最佳子结构性质（当字尾不相同时）

另一方面，当两字符串的字尾相同时，可通过短一些的最长相同子字符串获取。例如，当 X=<A, T, C, T, G, A>和 Y =<T, G, C, A, T, A>时，因为 X 和 Y 尾部的字母相同，所以可先找出 X^*=<A, T, C, T, G >和Y^*= <T, G, C, A, T>的最长相同子字符串<T, C, T>，再于尾部加上共同拥有的 A，就可以得到其最长相同子字符串为<T, C, T, A>，如图 3.6 所示。此时大问题的最优解（即<T, C, T, A>）一样包含小问题的最优解(即<T, C, T>)。

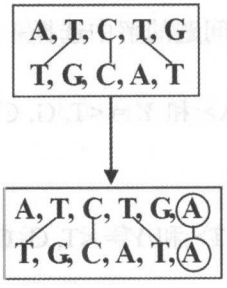


图3.6 最长相同子字符串问题的最佳子结构性质（当字尾相同时）

注意　此最佳子结构的性质确认了动态规划可以解决最长相同子字符串问题。

3.4.2　步骤2：利用数学式子描述大问题的最优解和小问题的最优解的关系

步骤1确认了大问题的最优解包含小问题的最优解，如此才有机会利用小问题的最优解组合拼凑出大问题的最优解。接下来，步骤2利用数学式子将此关系表达清楚。此式子也将在步骤3引导最后解的计算。

当考虑两个字符串 X 和 Y 的最长相同子字符串问题时，我们利用 $c[i,j]$ 代表并存储其相关小问题的最优解的长度。准确地说，$c[i,j]$ 代表字串 X 中前面 i 个字符(即$<x_1, x_2, \cdots, x_i>$)和字串 Y 中前面 j 个字符(即$<y_1, y_2, \cdots, y_j>$)所形成的最长相同子字符串的长度。

例如，当 X=<A, T, C, T, G, A>和 Y =<T, G, C, A, T, A>时，$c[2, 2]$ 记录<A, T>和<T, G>的最长相同子字符串的长度(即 $c[2, 2]=1$)；而 $c[3, 4]$ 记录<A, T, C>和<T, G, C, A>的最长相同子字符串的长度（即 $c[3, 4]=2$）。

当然，很容易可得 $c[i, 0]=c[0, j]=0$，因为其中的一个字符串是空的。当 X 和 Y 的字符串字尾相同时，我们可得 $c[i, j]=c[i\text{-}1, j\text{-}1]+1$(图 3.6 就是一个例子）。相对地，当 X 和 Y 的字符串字尾不相同时，$c[i, j]=\max(c[i\text{-}1, j], c[i, j\text{-}1])$（见图 3.5）。以上关系可以整理如下：

$$c[i,j]=\begin{cases}0, \text{if } i=0 \text{ or } j=0\\ c[i-1,j-1]+1, \text{if } i,j>0 \text{ and } x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]), \text{if } i,j>0 \text{ and } x_i\neq y_j.\end{cases}$$

这个数学式子再次指出 $c[i, j]$ 的值是可以利用3个小问题的值 $c[i\text{-}1, j\text{-}1]$、$c[i, j\text{-}1]$ 和 $c[i\text{-}1, j]$ 计算出来。这个关系可协助最后一个步骤的设计。

3.4.3 步骤3：设计算法，使得在计算大问题的最优解之前，所需要的所有小问题的最优解都事先被计算并存储了

最后，在设计动态规划算法时，需小心安排计算最优解的顺序，即当计算 $c[i,j]$时，其可能需要的 3 个值 $c[i\text{-}1,j\text{-}1]$、$c[i,j\text{-}1]$和 $c[i\text{-}1,j]$已事先被计算并存储起来了。如图3.7所示，当填写每个矩阵中的某一格时，需将其上方、左方和左上方的值先填好。

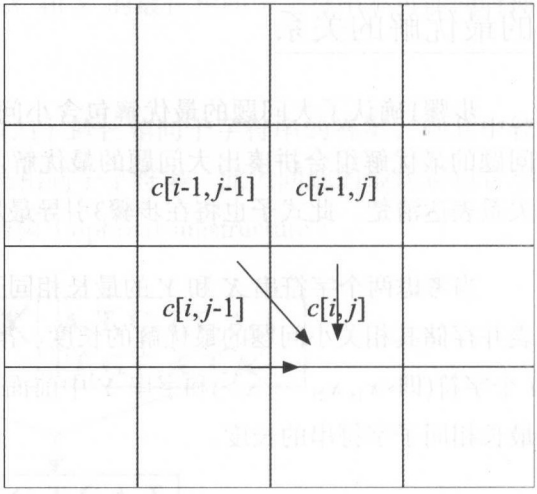


图3.7 计算 $c[i,j]$时，其所需的3个值 $c[i\text{-}1,j\text{-}1]$(左上方)、$c[i,j\text{-}1]$(左方)和 $c[i\text{-}1,j]$ (上方)需被事先计算出来。此图中的箭头指示出了计算最优解的顺序

因此，此算法可以按照从左到右和从上而下的顺序计算出所有的 $c[i,j]$，如图 3.8 所示。

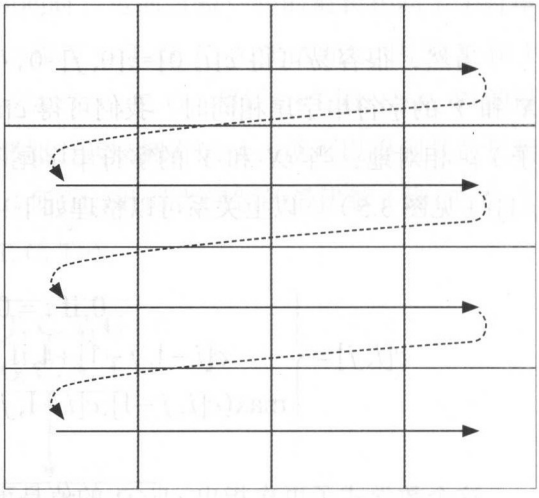


图3.8 动态规划算法须安排计算最优值的顺序

按照这个方法，当 X=<A, T, C, T, G, A, T>和 Y=<T, G, C, A, T, A>时的所有$c[i, j]$的值也可计算得出，如图 3.9所示。

		T	G	C	A	T	A
	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
G	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
T	0	1	2	2	3	4	4

图3.9 当 X=<A, T, C, T, G, A, T>和 Y =<T, G, C, A, T, A>时所有 $c[i, j]$ 的值

表 3.6列出了最长相同子字符串的动态规划算法算法。

表 3.6 最长相同子字符串的动态规划算法

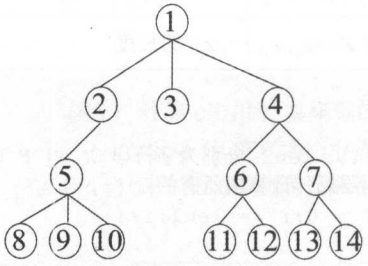
输入	两组字符串 X=<$x_1, x_2, \cdots, x_m$>，Y=<$y_1, y_2, \cdots, y_n$>
输出	最长相同子字符串 Z=<$z_1, z_2, \cdots, z_k$>的长度
步骤	

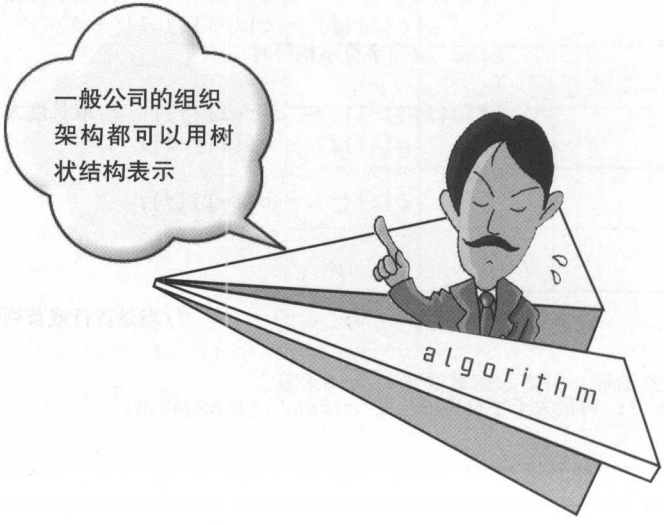
```
Algorithm lcs
{
Step 1: 令 len1, len2 分别为字符串 X 和 Y 的长度。
Step 2: 利用两层循环计算出所有的 c[i, j]。
        for(i = 0;i <= len1;i++)                    /*由上而下*/
        {
           for(j = 0;j <= len2;j++)                 /*由左至右*/
           {
             if(i != 0 && j != 0)
             {
               if(str1[i] == str2[j])               //当字符相等时
                      c[i][j] = c[i-1][j-1] + 1;
               else //当字符不相等时
               {
               if(c[i][j-1] >= c[i-1][j])           //取长度大者
                      c[i][j] = c[i][j-1];
               else
                      c[i][j] = c[i-1][j];
               }
             }
             else
               c[i][j] = 0;                         //当是首行或首列时，设置为0
           }
        }
Step 3: 将最大子字符串的长度 c[len1][len2]输出;
}
```

3.5　安排公司聚会

一个问题是否可用动态规划来解，有时不是一眼就能看出的。从表3.7来看如何进行此类的判断。

表 3.7　安排公司聚会问题

问题	老王要为公司员工安排一个非正式的聚会。公司的组织如同一棵树，董事长位于树根。为了使整个聚会成功，人事部门使用分数评估每位员工为聚会带来快乐的程度。另外，为使聚会有愉悦的气氛，员工和其直属上司不会同时参加。请设计一个算法，帮助老王安排聚会名单，使得此次聚会的快乐分数总和最大。
输入	矩阵 *happy*[*n*]代表 *n* 个员工产生快乐的分数，而矩阵 *children*[*n*, *n*]存储公司的树状组织架构，*children*[*i*, *j*]=1 代表 *i* 是 *j* 的直属上司。*n*=14 
输出	参与聚会的最优名单是{1, 2, …, *n*}的子集合，使得整体快乐分数总和最大



安排公司聚会问题可用动态规划来解吗？

应该如何判断？

应该先判断此问题是否有最优子结构性质。

什么是最优子结构性质？

检查是否大问题的最优解可以利用小问题的最优解组合得到。

若在公司找出 n 位员工是大问题，则小问题应该是什么？怎样的小问题的最优解对解大问题有帮助？

已知 n-1 位员工的小问题的最优解，好像对找到 n 位员工大问题的解帮助不大。

为什么？

因为公司的组织架构未被考虑，即员工和其直属上司不会同时参加聚会的特性。

公司的组织架构是什么？

就像一棵树（tree）。

一棵树的小问题可能是什么？

应该是小树或子树（subtree）吧！

哪些子树的解对找到大树的最优解有帮助？

嗯……

大树和子树的关系是什么？树的定义是什么？

由一个树根（root）和其子女（children）为树根的子树所组合而成。

3.5.1　安排公司聚会问题的最优子结构判断

以表3.7的公司组织为例，1号员工（即董事长）有两种可能，即“参加”和“不参加”聚会。

如果1号员工不参加，那么以2号员工为首的部门（即以2为树根的小树）的最优解（即2号员工的部门中最大快乐的聚会名单），配合以3号员工为首的部门（即以3为树根的小树）的最优解，再配合4号员工所带头的部门（即以4为树根的小树）的最优解，就是整个公司的最优解（最优聚会名单），如图3.10中3个虚线方块所示。

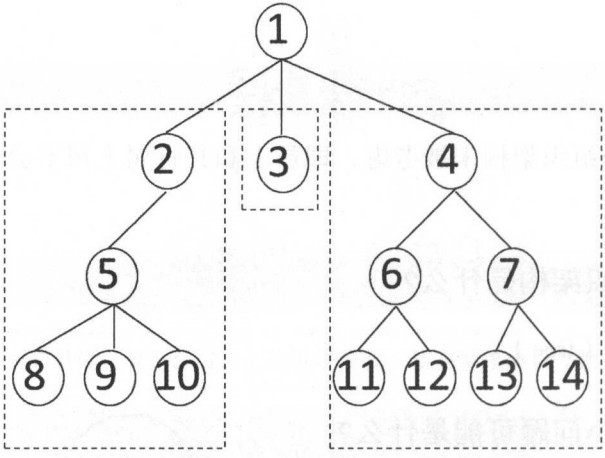


图3.10　当1号员工不参加聚会时的公司聚会问题

若1号员工参加，则 2、3、4 号员工不可以参加。但是此时1号可配合5号、6号和7号员工所带头的下属（即以5、6、7为树根的小树）的最优解，得到最优聚会名单，如图 3.11 所示。

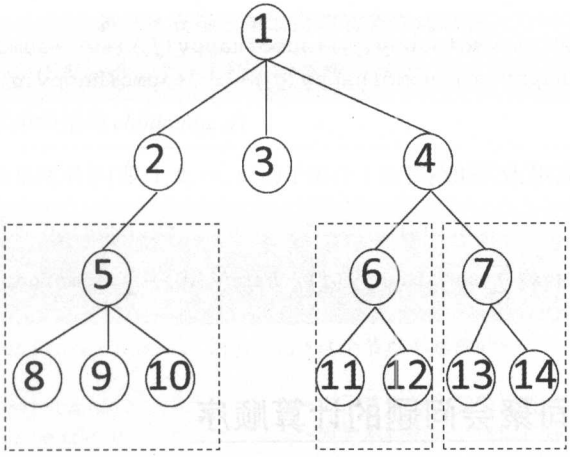


图3.11　当1号员工参加聚会时的公司聚会问题

从这两种（即1号员工参加或不参加）聚会名单的快乐分数中取分数高的，即可决定最优的聚会名单。

如此，我们就可以用小问题（以2、3、4、5、6、7为树根的小树）的最优聚会名单找到大问题（以1为树根的大树）的最优聚会名单了。因此，安排公司聚会问题具备最佳子结构性质，适合动态规划法求解。

3.5.2　安排公司聚会问题的递归关系

接下来进行下一个步骤：使用一个数学式子描述大问题的最优解和小问题的最优解的关系。

倘若以 v 为根的部门的最大快乐分数存储于 *sumofhappy*(v)中。当 v 是树叶（leaf）时，显然 *sumofhappy*(v)=*happy*(v)。

当 v 不是树叶时，必有一个以上的儿子。令其所有儿子为$\{j_1, j_2, \cdots, j_c\}$，并令 v 的所有孙子(即其所有儿子的儿子)为 $\{g_1, g_2, \cdots, g_d\}$。根据之前的讨论，下列式子可以描述大问题的最优解和小问题的最优解的关系。

$$sumofhappy(v)=Max\{sumofhappy(j_1)+sumofhappy(j_2)+\cdots+sumofhappy(j_c),\ happy(v)+sumofhappy(g_1)+sumofhappy(g_2)+\cdots+sumofhappy(g_d)\}。$$

上面的式子也可改写成：

$$sumofhappy(v)=Max\{\sum_{i=1}^{c}sumofhappy(j_i),\ happy(v)+\sum_{j=1}^{d}sumofhappy(g_j)\}。$$

3.5.3 安排公司聚会问题的计算顺序

进行最后的步骤：设计算法，使得在计算出大问题的最优解之前，其所需要的所有小问题的最优解都已被事先计算出。

首先，因为安排公司聚会问题考虑树状组织架构，需要一些变量存储树（公司的组织架构）的数据。令 v 为此公司的树状组织的内部节点(internal node)，则 T_v 代表以 v 为树根的树。以图 3.10 为例，节点 1 是整棵树的根（root），T_1 代表整个公司的大树。节点 2、3、4 是节点 1 的3个儿子，而 T_2 T_3 T_4 分别代表以 2、3、4 为树根的3棵小树。

用来描述这个算法的若干变量，说明如下：

- happy[v] 每一个节点 v 将其快乐指数存于 happy[v]中。
- sumofhappy[v] 以 v 为根的部门（小树）的最大快乐指数。
- children[v, j] 若 children[v, j]返回 true，则代表节点 v 为节点 j 的父节点。

表3.8中的算法可以计算出最优聚会名单的快乐指数总和，并存入 *sumofhappy*[1] 中。只要稍加修改，即可找出最优聚会名单。

表 3.8 安排公司聚会的动态规划算法

输入	矩阵 *happy*[*n*]代表 *n* 个员工产生快乐的分数 公司的树状组织架构 *children*[*n*, *n*]
输出	参与聚会的最优名单(即{1, 2, …, *n*}的子集合）的整体快乐分数 *sumofhappy*[1]
步骤	见下方代码

```
Algorithm getSum(int node)  /*node 的初始值为 1*/
{
for(i = 1;i <= N;i++)       /*检查是否此点是一个树叶*/
{
    if(children(node, i))   /*这个节点不是树叶*/
    {
       getSum(i);           /*将子节点作为父节点处理*/
       isLeaf= 0;           /*因为不是树叶，所以标成 NULL*/
    }
}

    if(isLeaf)                          /*此点是树叶时*/
    {
    sumofhappy[node] = happy[node];     /*记录树叶的快乐分数*/
    }
    else                                /*当此点不是树叶时*/
    {
       sum2=happy[node];
       for(i = 1;i <= N;i++)
       {
          if(children(node, i))
          /* 若此点是某一点的父节点，则要将其值加入 sum1 */
          {
              sum1 += sumofhappy[i]; /*将子节点的分数加总*/
              for(j = 1;j <= N;j++)
              if(children(i, j)) sum2 += sumofhappy[j];
          }
       }
    if(sum1 >= sum2)                    /*获取最高的快乐分数*/
          sumofhappy[node] = sum1;
    else
          sumofhappy[node] = sum2;
    }
}
```

万一董事长不可以参加公司聚会，这样好吗？

怎么做，董事长一定会参加又不违反公司规定？

请人事部门调高董事长（带给员工）的快乐分数就好了。

这样会不会太假了！

太棒了，就这么办。

3.6　动态规划的技巧

如何判断一个问题是否可用动态规划来解？

本章的4个范例指示我们使用下列步骤思考，以及使用动态规划的技巧。

（1）判断大问题的最优解是否可以利用（多个）小问题的最优解组合并解出。

（2）若可以，则尝试写出大问题最优解和小问题最优解之间的递归关系。

（3）根据上述递归关系设计算法。先计算小问题的最优解，再计算出大问题的最优解。

哪种问题使用动态规划特别有效率？

动态规划有什么优点？

什么是动态规划？

计算并存储小问题的解，并将这些解组合成大问题的解。

什么是动态规划的特色？尤其是在加速计算方面的优点。

计算过的小问题不会重复计算。

如此有什么好处?

可以避免无谓的重复计算，以加快算法求解时所需的时间。

怎样可充分发挥动态规划的优点?

如果计算过小问题可被很多大问题利用来求解，也许更可以发挥出动态规划的优点。

怎样可以知道一个小问题的解可用于组合多少个大问题的解?

我好像写过两者之间的关系。对了，大问题最优解和小问题最优解之间的递归关系式子。

最后列出一些可以被动态规划解的问题，以供读者参考。

（1）矩阵链相乘（matrix-chain multiplication）：多于3个矩阵连乘时，不同顺序的矩阵相乘所得的结果相同，但需要的乘法运算总次数不同（一般加减法较省时间，故在此不计算）。此问题要找到最少乘法运算的矩阵相乘顺序。

（2）最优多边形三角分割（optimal polygon triangulation）：将一个多边形分割成多个三角形（分割点需在顶点上，且分割线不可相交），使其所需的分割线段长度总和最小。

（3）任意两点间的最短路径（all-pairs shortest-paths）：计算一个图中任意两个顶点（vertex）之间的最短路径，此图中不含有总和为负值的回路（cycle）。

（4）最优二叉搜索树（optimal binary search tree）：将被搜索的值置于二叉树（binary tree）的树叶（leaf）上。假设已经知道每个被查询的值的概率，此问题是创建一棵二叉搜索树，使得其所需的平均搜索时间最短。

（5）资源分配问题（resource allocation problem）：考虑m项资源和n个计划，当某项资源被分配到某个计划时，会有相对应的利益产生。此问题是找到一个资源分配的方式，使得整体的利益最大。

学习效果评测

1. 最长严格递增子字符串：编写一个程序，从一串整数中找出最长严格递增子字符串（longest strictly increasing subsequence），即从一串整数中找到一个子集合；按照原来的顺序排列时，下一个数比上一个数值大，并且此子集合个数最大（即排出来的子字符串最长）。

```
输入：
-17 11 9 2 3 8 8 10

输出：
-17 2 3 8 10
```

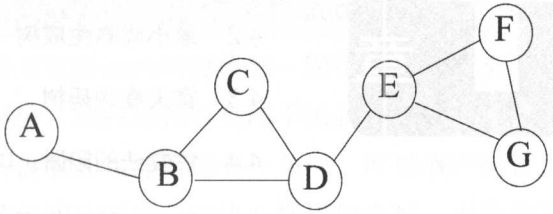
2. 给一个二维整数矩阵，找出一个子矩阵（sub-rectangle），该矩阵内所有数字的总和最大。例如，下列矩阵的解是下方的子矩阵：

```
-1  0   -9
-2  3   14
-2  4   20


3   14
4   20
```

而且其总和为 41。

```
输入：
3 3                    (矩阵的行数和列数)
-1 0 -9                (以下是矩阵的数据)
-2 3 14
-2 4 20

输出：
41                     (子矩阵最大总和)
```

3. 一台文件服务器（file server）希望被布置于一个连通（connected）网络的中心，使得网络上其他设备可以用最少的步数（hop）得以连接此服务器。例如，当此服务器被置于A时，最远的G最少需要花 4 步连接到A。但是当此服务器被置于D时，最远的F只需要两步即可抵达D。那么点D就是网络的中心点。



输入：

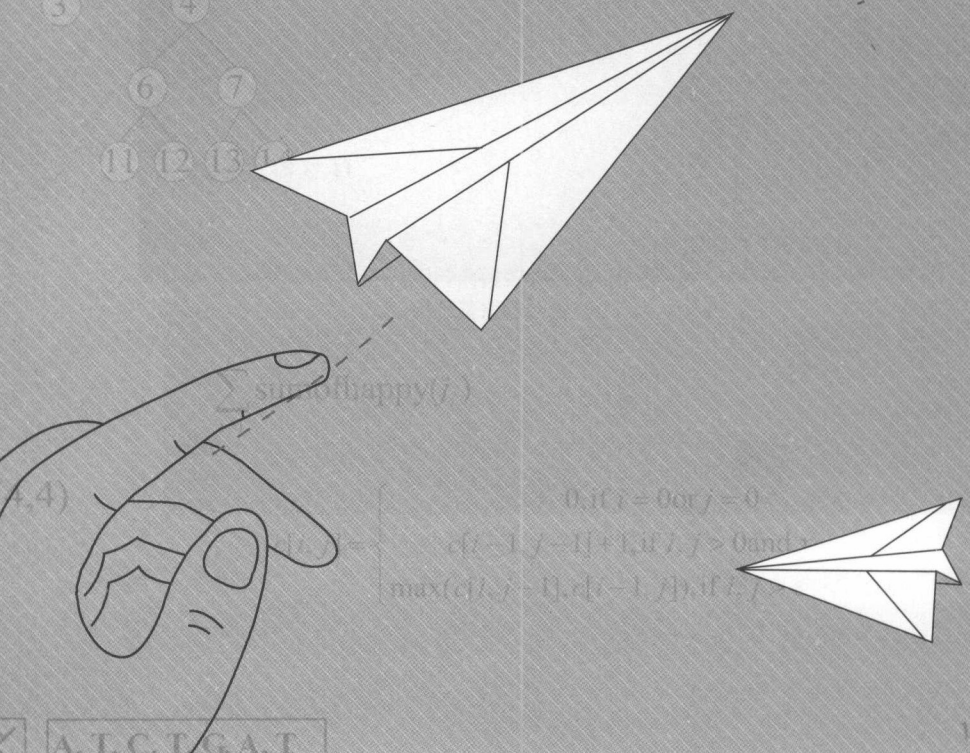
```
6                (网络上点的个数，并且以1，2，3……编号)
9                (网络上线的条数)
1 2              (以下为每一条网络线的数据)
1 4
2 3
2 4
2 5
3 4
4 5
4 6
5 6
```

输出：

```
4                (网络中心点的编号)
```

第4章 贪婪法

章节大纲



4.1　何谓贪婪法

什么是贪婪法（greedy method）？

简而言之，就是重复地（或贪婪地）根据一个法则挑选解的一部分。当挑选完毕时，最优解也就出现了。

印度有一位农夫在田地的河边捡到一颗很漂亮的石头。他将石头带回家给小孩玩，小孩玩腻后，就将那颗石头随手丢到杂物堆的角落。有一天，一位珠宝商路过他家，告诉农夫在此附近有一条河，河里盛产钻石。农夫心想，种了一辈子的田太辛苦又赚不到钱，就决心把田地卖掉，去寻找那条产钻石的河。农夫找了好多年，无功而返。有天闲来无事整理家里时，在杂物堆里发现了那颗以前在河边捡来的石头，这才发现那竟是颗价值连城的大钻石。而他多年来苦心寻找的钻石河，正好位于他卖掉的田地内。

农夫寻找宝石的故事暗示：眼前的东西有时是最珍贵的。贪婪法一旦使用得当，会是一个有效率的策略，即便贪婪法也许无法解决所有问题。

4.2　最小成本生成树

第一个例子是使用最小成本架设网络的问题，如表4.1所示。

表 4.1　最小成本网络架设问题

问题	有一位网络工程师替一家公司规划架设一个网络。他想将所有网络设备连接成一个完全连通的网络，但是希望所使用的网络成本可以降到最低。请替他设计一个算法解决此问题
输入	一个网络可能的构建图(graph) $G=(V, E)$，其中顶点（vertex，也称为节点或点，本书统一称为顶点）代表网络设备，顶点和顶点之前的连线（edge，也称为边，本书大多数情况下都称为连线）代表两台网络设备可以连通。连线上标注的整数代表连接这两台网络设备所需的架设成本
输出	将全部网络设备连接成一个连通的网络时，所需最小成本的连接方式

这个问题是在输入的图（graph）中寻找怎样的答案？

应该是寻找此图的一部分，即子图（sub-graph）。

任意子图都可以吗？

不！需要将所有顶点连接在一起的子图。

还有其他条件吗？

这个子图的成本总和（sum）还必须最小。

上面描述了最小成本生成树（minimum cost spanning tree）问题：在一个有权重（weight）的图中，寻找一个子图（即此图的子集合）符合下列条件：

（1）连接在一起。　[此为树（tree）的性质]

（2）经过每一个顶点。　[此为生成（spanning）的性质]

（3）连线的成本总和最小。　[此为最小成本（minimum cost）的性质]

如图4.1所示，粗线的子图为整个图中一个最小成本生成树。

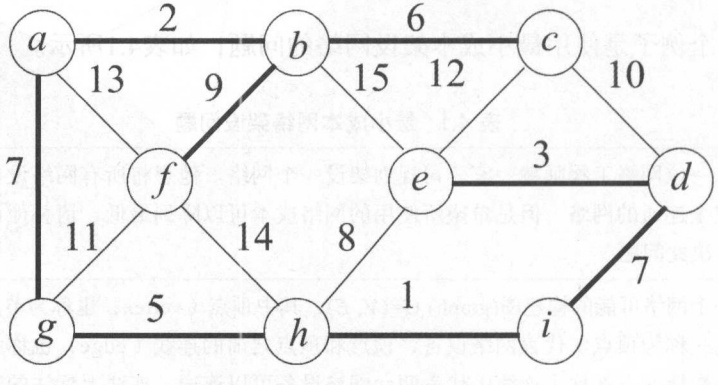


图4.1　一个最小成本生成树（粗线的子图）

在任意一个连通图（connected graph）中找出其最小成本生成树后，会发现此树不含回路，而且任意地额外加入一条新的连线于此树上，就会产生一个唯一的回路。如图4.1所示，加入线（e, h）即可产生回路{dehi}。相反地，一旦从这棵生成树上任意地删除一条连线，就会造成整棵树不连通。用构建一个连通网路的问题打比方，每一棵最小成本生成树是将网络连在一起的最精简（最省线材）的方式。

如何在一个图中找到一棵最小成本生成树?

嗯！不知道。

观察最小成本生成树的例子(见图4.1)，试着找这些最优解的特点?

一棵生成树是通过每一个顶点且拥有连接、成本最小及无回路的特点。

利用这些特色可以找到最小成本生成树吗?

嗯！好像不行。

试着比较落在最小成本生成树中的连线和不在其中的连线的差别?

看不出来有什么差别，除了任意地额外加入一条连线在生成树上就会产生唯一的回路。

注意此回路，试着找出生成树上的连线和新加入的连线之间有什么差别?

似乎在此回路中新加入（即不在最小成本生成树上）连线的成本都比回路中其他（即在最小成本生成树上）连线的成本高。

这个发现有助于找到一棵最小成本生成树?

也许可以先不考虑成本较大的连线，或按照连线的成本从小到大的顺序构建一个连通网络。

试几个例子看看，可否找到一棵最小成本生成树？

有些例子在连接的过程中会发生回路。这种情况需要避免，因为目标是要找到一棵树，而树是无回路的。

目前你有什么想法？

按照连线的成本从小到大的顺序连通整个网络，同时需要避免发生回路。

Kruskal 的最小成本生成树的算法是：按照连线的成本，从小到大按序选择连线连通整个网络，但是在此过程中需避免发生回路。以图 4.1 为例，此算法的执行过程如图4.2所示。

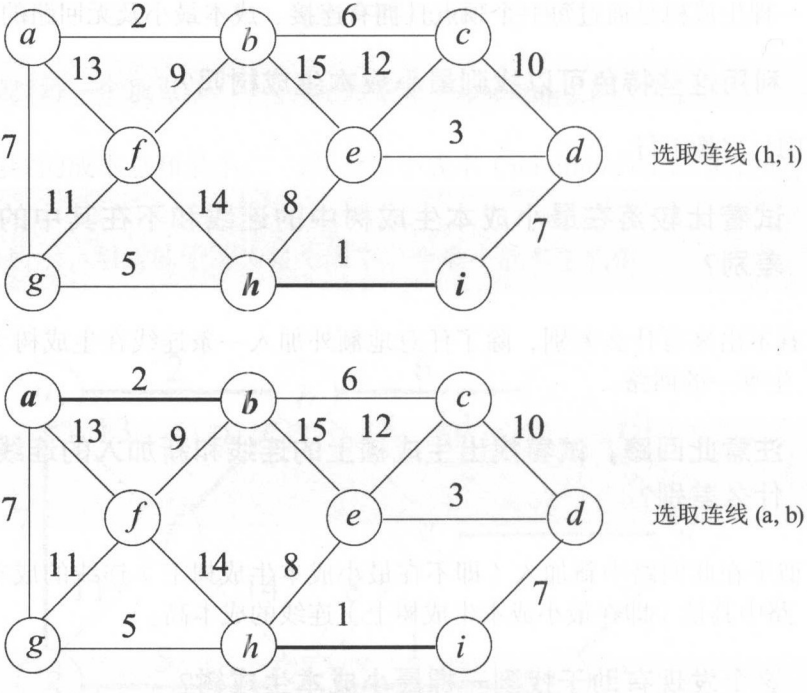
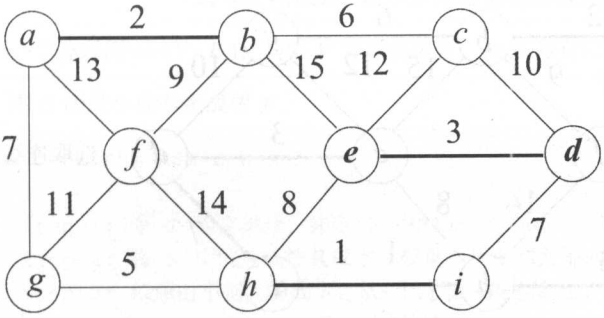
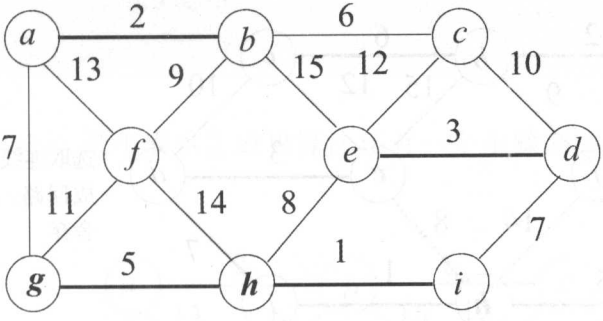


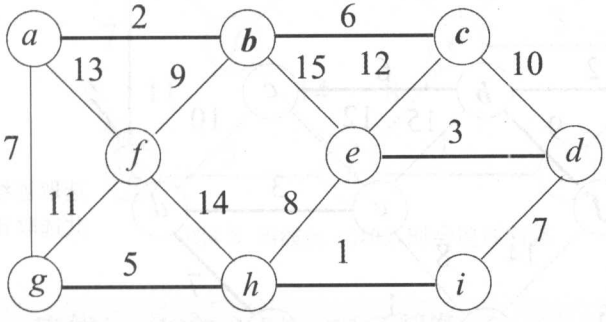
图4.2 Kruskal 最小成本生成树算法的执行范例



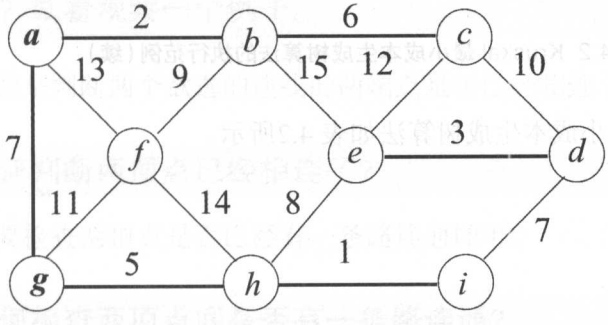
选取连线(e, d)



选取连线 (g, h)



选取连线 (b, c)



选取连线 (a, g)

图4.2 Kruskal 最小成本生成树算法的执行范例(续)

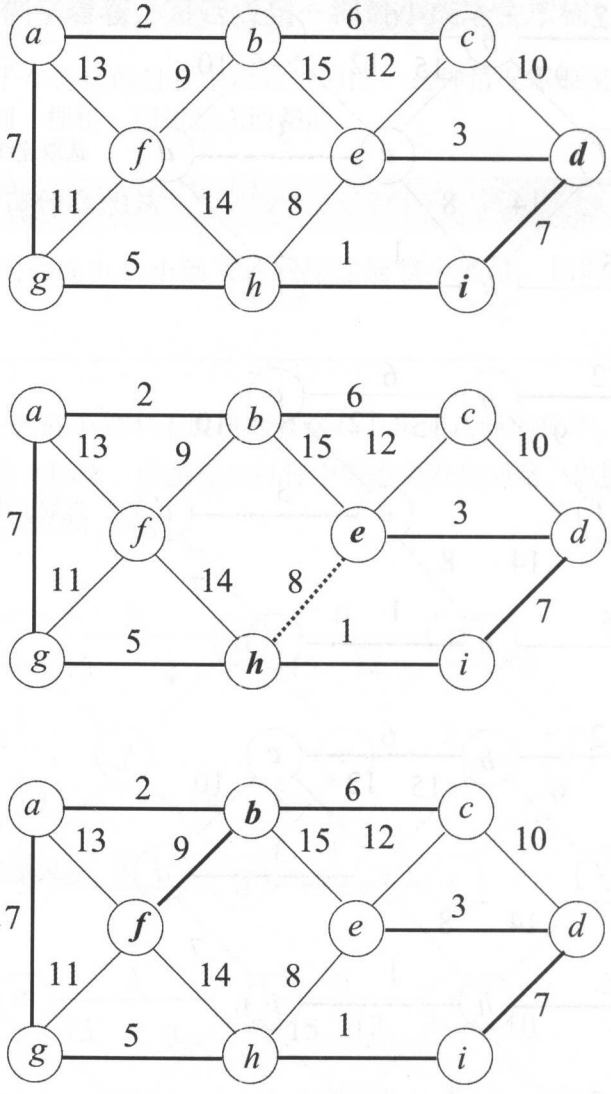


图4.2 Kruskal 最小成本生成树算法的执行范例（续）

详细的 Kruskal 最小成本生成树算法如表 4.2所示。

表 4.2 Kruskal 最小成本生成树的算法

输入	一个连通图 $G=(V, E)$，其中 V 为顶点集合，而 E 为有权重的连线集合
输出	图 G 的最小成本生成树 T

步骤	Algorithm MST-Kruskal
	{
	Step 1: 令 T 为空集合，并令 V 中的每一个顶点 v 为一个集合。
	Step 2: 将 E 中的连线按其成本（权重）从小到大排列。
	Step 3: 依照由小到大的成本，从E中选取一条连线(u, v)，并执行下列指令：{if $T \cup \{(u, v)\}$不会形成回路，Then将(u, v)加入 T 中。}直到选中$\|V\|-1$条连线为止。
	}

Kruskal 最小成本生成树算法还有一个步骤需要进一步讨论，如图4.3所示。

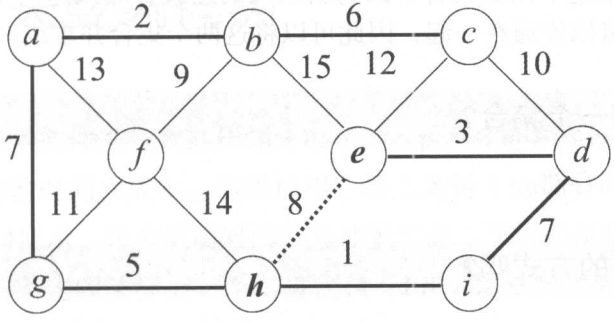


图4.3 当线(e, h)加入时会造成回路

当一连线(e, h)加入时，如何判断 $T \cup \{(e, h)\}$会不会形成回路？试着观察一个例子。

好像是判断两个欲选的连线的两端点是否已经相连了。

如何判断两顶点已经相连了？

只要检查两顶点是否已经有一条路连通即可。

如何检查两顶点间是否有一条路连通？

嗯……

所有连通的点有什么共同性质？

所有连通的顶点彼此互相连接。

如何记录这种关系？

将所有相连的顶点用一个集合记录。

这个关系什么时候会被改变？

当加入的新连线连通两个不同的集合时。

此时关系会有怎样的改变？

这两个原来是不同的集合，因为新加入的连线，导致这两个集合的所有顶点都彼此可以连通在一起，因此可以将这两个集合并集成一个大的集合。

如何存储一个集合？

可以用矩阵存储。

还有更好的方式吗？

嗯……

为何选中 |V|-1 条连线后，此算法即可停止？

因为已经找到一棵生成树了。

$|V|$ 个顶点的生成树有多少条连线？

好像是 $|V|-1$。

这个算法是对的吗？其时间复杂度为多少？

嗯……

Kruskal 最小成本生成树算法的时间复杂度为O (|E| log |E|)，如表4.3所示。注意Step 3 需运用较有效率的集合并集（union）和查找（find）的数据结构。

表 4.3 Kruskal 最小成本生成树算法的时间复杂度

指令	执行次数
Step 1: 令 T 为空集合，并令 V 中的每一个点 v 为一个集合	$O(\|V\|)$
Step 2: 将 E 中的连线按其成本从小到大排列	$O(\|E\| \log \|E\|)$
Step 3: 按照从小到大的成本，从 E 中选取一连线(u, v)，并执行下列指令: {if $T \cup \{(u,v)\}$不会形成回路，Then 将(u, v)加入 T 中。} 直到选中 |V|-1 条连线为止	$O(\|E\|)$
时间复杂度	$O(\|E\| \log \|E\|)$

4.3 霍夫曼编码树

若将一份文本文件进行数据压缩（data compression）后，再上传到网络上，可以减少传输时间和成本。编码树用一棵二叉树（binary tree）表示编码的方法。如图4.4所示，每个被编码的符号被置于此二叉树的树叶（leaf）上，而且树上的每一条连线被标上一个二进制位（bit）的 0（向左的连线）或 1（向右的连线）。

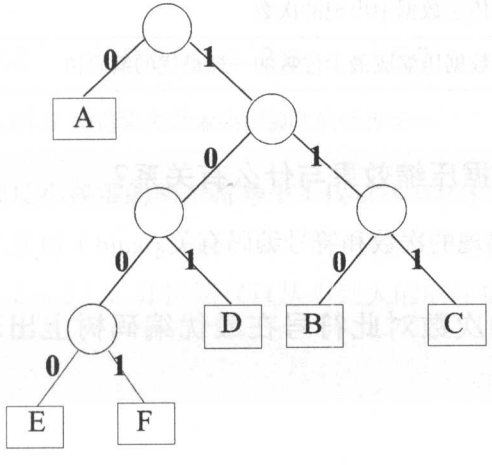
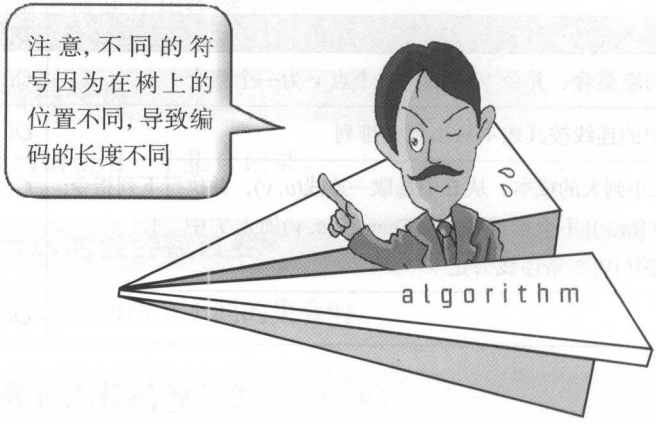


图4.4 编码树

在一棵编码树上，从树根（root）走到一个特定的树叶（leaf）会形成一条唯一的路径。收集这条路径各个连线上的 0 与 1 数字串，也就是此树叶上的符号所对应的编码。例如，图4.4 中的 F对应 1001，而 D 对应 101。



不同的编码树会有不同的编码方式，也会产生不同的数据压缩效果。如果知道每个符号在传送数据中出现的次数，如何构造出一棵最优的编码树将传送的数据压缩成最少的位，就成为一个值得探讨的问题，如表4.4所示。霍夫曼编码（Huffman code）就是其中一种方法。

表 4.4 构造最优的编码树

问题	已知每个符号在传送数据中出现的次数。请设计一个程序找出一棵最优的编码树，使得使用此树的编码方式可以将传送的数据压缩成最少的位
输入	每个符号在传送数据中出现的次数
输出	可将传送的数据压缩成最少位数的一棵最优的编码树

编码树的数据压缩效果与什么有关系？

应该与符号出现的次数和符号编码有关。

符号出现的次数对此符号在最优编码树上出现的位置有何影响？

嗯……

出现最多次数的符号应该放在编码树的什么地方？

当然是越靠近树根（root）越好！因为这样对应的位数总和会越少。

出现最少次数的符号应该放在编码树的什么地方?

越靠近树叶（leaf）越好，这样可把靠近树根的位置留给出现较多次数的符号。

按此想法，如何构造一棵最优的编码树?

根据符号出现的频率安排出现在树上的位置：即在树的上层放置常出现的符号，在树的下层放置较少出现的符号。

一开始要怎么做?

也许可以将最少出现的两个符号放在树的最下层。

霍夫曼编码树就是贪婪地以“出现最少次数的两个符号放在树的最下层”的方法所构造出的一棵最优的编码树。下面使用一个范例说明。

步骤 1 一开始整个集合包含所有符号，将每个符号的权重设置成其出现的次数，并且按照权重从小到大排列，如图4.5所示。

图4.5 构造霍夫曼编码树算法的过程之一

步骤 2 将集合中出现最小权重的两个符号 E（权重 1）和 F（权重 2）取出，合并成一棵二叉树（binary tree）后，将此树的权重设为 E 和 F 权重的和（即 1 + 2 = 3），并按照权重从小到大的顺序插入原顺序中，如图4.6所示。

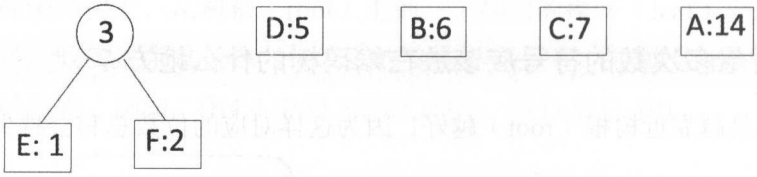


图4.6 构造霍夫曼编码树算法的过程之二

步骤 3　将集合中出现最小权重的树（权重 3）和 D（权重 5）取出，合并成一棵二叉树。将此新树的权重设为 8（= 3 + 5），并按照权重从小到大插入原顺序中，如图4.7所示。

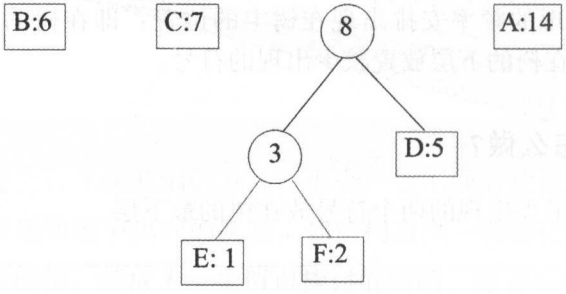


图4.7 构造霍夫曼编码树算法的过程之三

步骤 4　将集合中出现最小权重的 B（权重 6）和 C（权重 7）取出，合并成一棵树后，并将此新树的权重设为 13（= 6 +7），再次放回原顺序中，如图4.8所示。

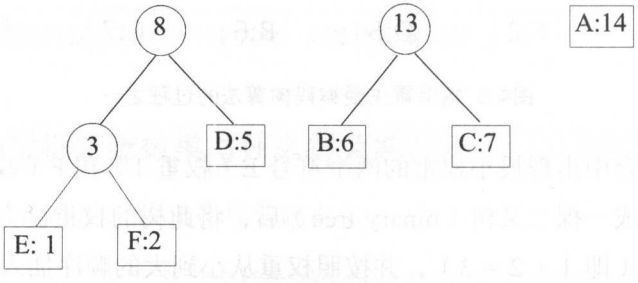


图4.8 构造霍夫曼编码树算法的过程之四

步骤 5 将集合中出现最小权重的树（权重 8和权重 13）取出，合并成一棵树（其权重设为 21）后，依次放回原集合，如图4.9所示。

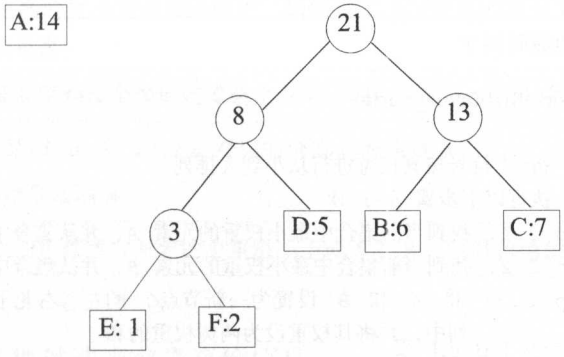


图 4.9 构造霍夫曼编码树算法的过程之五

步骤 6 将集合中剩下的两个字符合并成一棵树后，放回原集合。此新树的权重为 35（= 14 + 21），如图4.10所示。

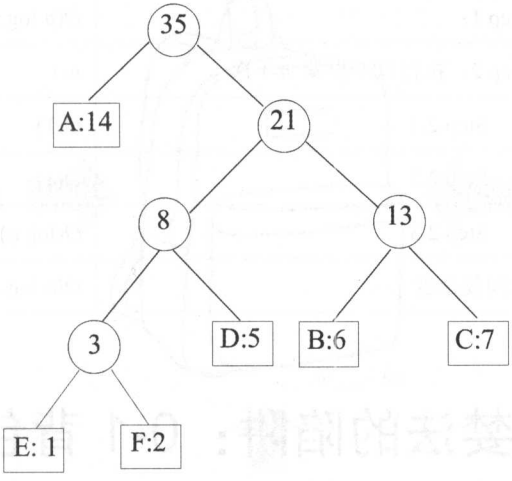


图4.10 构造霍夫曼编码树算法的构造结果

构造霍夫曼编码树的贪婪算法如表4.5所示。

表 4.5 构造霍夫曼编码树的算法

输入	n 个符号及其权重（每个符号出现的次数）
输出	一棵最优的编码树 T

步骤	Algorithm Huffman_code
	{
	Step 1: 将n个符号按其权重进行从小到大排列
	Step 2: 执行以下步骤 n-1 次。
	Step 2.1: 找到当前集合中最小权重的元素 A，并从集合中删去元素 A。
	Step 2.2: 找到当前集合中最小权重的元素 B，并从集合中删去元素 B。
	Step 2.3: 将 A 和 B 设置为一新节点 C 的左、右儿子。将新树加入原排列中，并将其权重设为两树权重的和。
	}

构造霍夫曼编码树的时间复杂度分析如表4.6所示。

表 4.6 构造霍夫曼编码树的时间复杂度分析

指令	执行次数
Step 1：	$O(n \log n)$
Step 2：执行以下步骤 n-1 次	n-1
Step 2.1	$O(1)$
Step 2.2	$O(1)$
Step 2.3	$O(\log n)$
时间复杂度	$O(n \log n)$

4.4 贪婪法的陷阱：0-1 背包问题

当贪婪法使用得当（如前两节所述）时，会是一个有效率的策略。但是，目前并非所有问题都可用贪婪法解决。下面介绍的 0-1 背包问题（0-1 Knapsack problem）就是一个例子，如表4.7所示。

表 4.7 0-1 背包问题

问题	超市促销举办限时抢购活动，被选中的客户可以在规定的时间内用一个袋子尽量装超市货架上的商品。超市的商品有值钱的和不值钱的，客户想把值钱的东西都带走，但是装东西的袋子有重量的限制。请问在不超过重量的限制下，客户该带走哪些商品总价值最高？ 注意，每件商品必须整个被取走或整个留下，如同只能从 0 和 1 中做选择一样
输入	（1）n 个商品$\{d_1, d_2 \cdots, d_n\}$及 d_i 对应的价值 v_i 及重量 w_i （2）袋子的重量限制 W
输出	一个商品列表使得列表内对象的总重量小于或等于 W，且其价值的和为最大

直觉上，贪婪地挑选当前最有价值且轻的商品，会有最优解，也就是从大到小按照{价值/重量}的值挑选。表4.8就是一个0-1背包问题的例子，表4.8的解析如图4.11所示。

表 4.8 0-1 背包问题的一个范例

	d_1	d_2	d_3
重量 w_i	10	20	30
价值 v_i	60	100	120
价值/重量	6	5	4

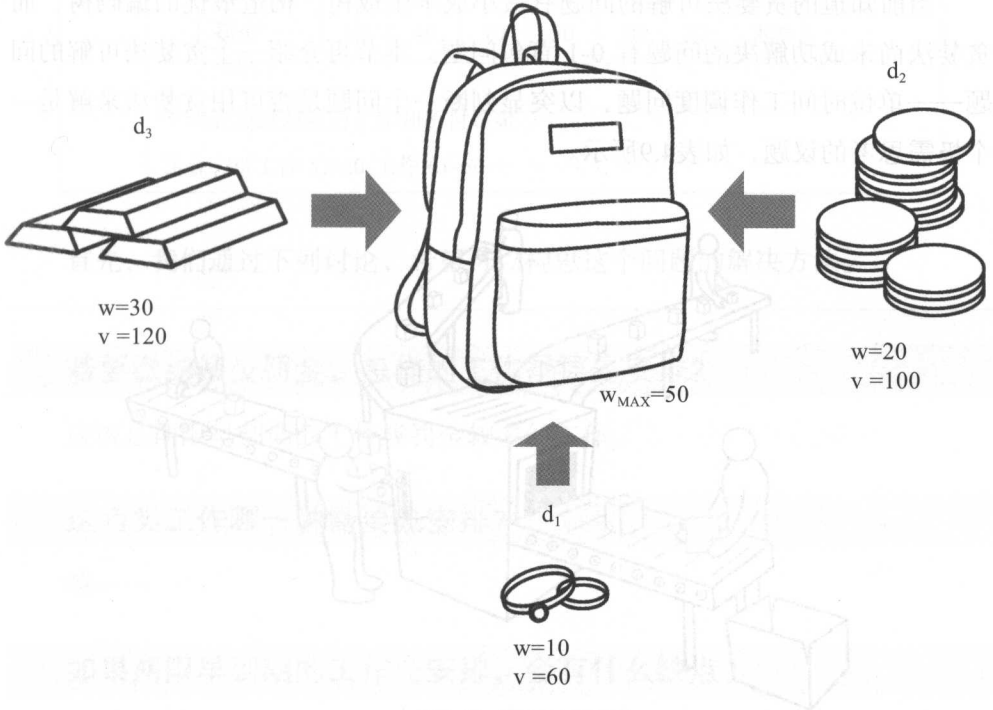


图4.11 表4.8的解析

令袋子的重量限制 W=50。若按照从大到小{价值/重量}的值挑选，则选入袋子的商品为 d_1 和 d_2，而总价值为 60 + 100 = 160。注意，此刻 d_3 不可同时纳入，否则总重量为 60，就超过袋子的负荷(W=50)了。但可惜的是，此解并非最优。若取 d_2 和 d_3，则总价值为 100+120=220，且总重量为 20+30=50，并未超过袋子的负荷。

从上例可知，使用贪婪法未能成功地解答 0-1 背包问题。当每件商品可任意被取走一部分时（如取走商品 d_1 的35%），这类背包问题被称为部分背包问题（fractional knapsack problem）。有趣的是，部分背包问题可以使用上述贪婪法找到最优解。注意，当最后取一件完整商品会超过袋子的负荷限制时，此法只取这种商品的一部分，使得整个袋子装满即可。

4.5　单位时间工作调度问题

当前知道的贪婪法可解的问题有最小成本生成树、构造最优的编码树，而贪婪法尚未成功解决的问题有 0-1 背包问题。本节再介绍一个贪婪法可解的问题——单位时间工作调度问题，以突显判断一个问题是否可用贪婪法来解是一个极需思考的议题，如表4.9所示。

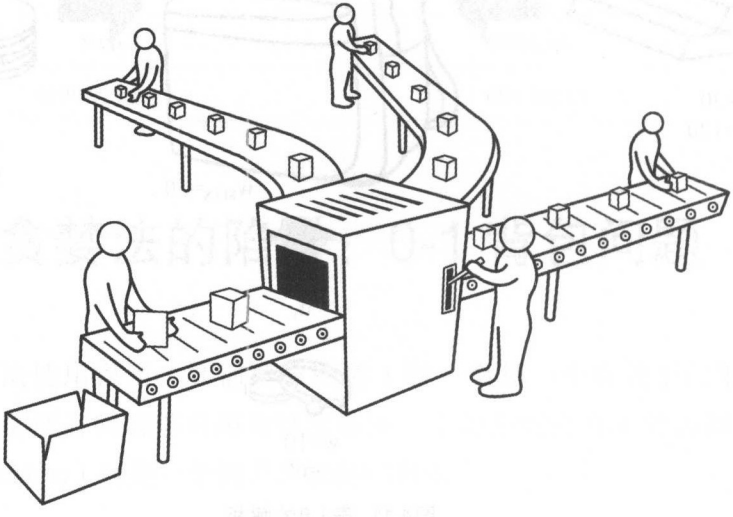


表 4.9 单位时间工作调度问题

<table>
<tr><td>问题</td><td>老王的工厂只有一台机器，但需完成的工作有n件，每份工作需占用这台机器一日的工作时间。每份工作都有完成的期限（deadline），一旦未在期限内完成，就需缴纳罚金（penalty）。请帮老王编写一个程序完成工作调度，使他缴纳最少的罚金</td></tr>
<tr><td>输入</td><td>n 件工作$\{J_1, J_2, \cdots, J_n\}$及 J_i 对应的完成期限为 D_i，未完成时需缴的罚金为 P_i
<table>
<tr><td>工作编号</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>期限</td><td>4</td><td>2</td><td>4</td><td>3</td><td>1</td><td>4</td><td>6</td></tr>
<tr><td>罚金</td><td>70</td><td>60</td><td>50</td><td>40</td><td>30</td><td>20</td><td>10</td></tr>
</table></td></tr>
<tr><td>输出</td><td>找出需缴纳罚金最少的工作调度
<table>
<tr><td>工作编号</td><td>2</td><td>4</td><td>1</td><td>3</td><td>7</td><td>5*</td><td>6*</td></tr>
<tr><td>排程日期</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>期限</td><td>2</td><td>3</td><td>4</td><td>4</td><td>6</td><td>1</td><td>4</td></tr>
<tr><td>罚金</td><td>60</td><td>40</td><td>70</td><td>50</td><td>10</td><td>30*</td><td>20*</td></tr>
</table>
*代表超过期限的工作和所付罚金
罚金=30(工作 5)+20(工作 6)=50</td></tr>
</table>

首先，我们通过下列讨论，希望可以构思这个问题的解决方法。

若要缴纳最少罚金，怎样的工作需要先安排？

应该是期限早到期的工作或罚金较多的工作。

这两类工作哪一种需要先安排？

嗯……

如果期限早到期的工作先安排，会有什么缺点？

应该是当罚金较多的工作想要先安排时，却被期限早到期的工作占用了机器。”

相反地，如果罚金较多的工作先安排，会有什么缺点？

应该是当期限早到期的工作想要先安排时，却被罚金较多的工作占用了机器。

回到原来的问题，这两类的工作哪一种需要先安排？

嗯……

调度的目的是什么？

找到缴纳罚金最少的调度。

哪一种调度最有可能缴纳较少罚金？

嗯……

每一种调度下，被处罚的工作罚金的情况是怎样的？

我想一下。第一种调度当罚金较多的工作想要先安排时，却被期限早到期的工作占用了机器，罚的是罚金较多的工作；相对地，第二种调度当期限早到期的工作想要先安排时，却被罚金较多的工作占用了机器，罚的是期限早到期的工作。

哪一种调度最有可能缴纳较少罚金？

嗯，也许应该先不用第一种调度。不必先承受罚金较多的处罚，如果使用第二种调度，罚的就是期限早到期的工作，也许缴的罚金比较少。

将罚金较多的工作先安排的方法可以设计出一个贪婪算法。下面使用一个范例说明。

步骤 1 将所有工作按照罚金的大小排列好。

工作编号	1	2	3	4	5	6	7
罚金	70	60	50	40	30	20	10

步骤 2 按照 Step 1 所得的顺序，将每件工作一一运用下列步骤判断是否加入当前的工作调度中。

步骤 2.1 将工作 1 排入排程日期。

工作编号	1						
排程日期	1						
期限	4						
罚金	70						

步骤 2.2 试着将工作 2 排入排程日期，因其期限为 2（比工作 1 的期限 4 早），故将工作 2 排于工作 1 之前。

工作编号	2	1					
排程日期	1	2					
期限	2	4					
罚金	60	70					

步骤 2.3 工作 3 和工作 4 排入排程日期时，也按其期限先后排入。

工作编号	2	4	1	3			
排程日期	1	2	3	4			
期限	2	3	4	4			
罚金	60	40	70	50			

步骤 2.4　若要将工作 5 排入排程日期，则因其期限为 1，需将前面的工作 2、4、1 及 3 的排程日期都向后推一日，但会造成工作 3 超过期限。因此工作 5 先不排入（预定被罚）。

工作编号	5*	2	4	1	3		
排程日期	1	2	3	4	5		
期限	1	2	3	4	4		
罚金	30*	60	40	70	50		

步骤 2.5　若要将工作 6 排入排程日期，则因其期限为 4，可排在工作 3 之后，但会造成工作 6 超过期限。因此工作 6 也暂时不排入（预定被罚）。

工作编号	2	4	1	3	6*		
排程日期	1	2	3	4	5		
期限	2	3	4	4	4		
罚金	60	40	70	50	20*		

步骤 2.6　工作 7 顺利排在工作 3 之后。

工作编号	2	4	1	3	7		
排程日期	1	2	3	4	5		
期限	2	3	4	4	6		
罚金	60	40	70	50	10		

步骤 2.7　最后将先前未排的工作 5 和 6 任意排入排程日期，并接受处罚。

工作编号	2	4	1	3	7	5*	6*
排程日期	1	2	3	4	5	6	7
期限	2	3	4	4	6	1	4
罚金	60	40	70	50	10	30*	20*

单位时间工作调度问题的贪婪算法步骤如表4.10所示。

表 4.10 单位时间工作调度问题的贪婪算法

输入	n 件工作$\{J_1, J_2, \cdots, J_n\}$及 J_i 对应的完成期限为 D_i，未完成时需缴的罚金为 P_i
输出	找出需缴纳罚金最少的工作调度

步骤	Algorithm Unit_Time_Job_Scheduling
	{
	Step 1: 按照罚金从大到小，将所有工作排序：$\{J_1, J_2, \cdots, J_n\}$。
	Step 2: J_1 纳入排程 $S=\{J_1\}$。
	Step 3: for i=2 to n do
	{
	if 所有的工作 SU{J_i} 因为 Ji 的加入，都可以在期限前完成，then S = SU{J_i}（上述判断可将工作 SU{J_i} 按照其期限从早到晚排列，并检查是否有工作超出期限）。
	}
	Step 4: 将在 Step 3 未被排入的工作任意安排在其他工作调度的后面。
	}

4.6　证明贪婪法并介绍Matroid理论

单位时间工作调度问题的贪婪算法是对的吗？

老师说的一定正确。

Kruskal 最小成本生成树算法是对的吗？

嗯，Kruskal挺有名的，他设计的算法应该差不了多少。

有没有方法证明这些贪婪法是正确的呢？

嗯……

本节所介绍的理论可用于证明一些贪婪法是正确的。贪婪算法的设计常需要提供证明，而证明需要严格的论证，总是令人不易接受。第 4.2 节中介绍的Kruskal最小成本生成树算法的证明如下，你可以试试看此证明正确与否。

Kruskal 算法在一连通图中找到最小成本生成树

证明：

假设由 Kruskal 算法找到的树是 t，而 t' 是一棵最小成本生成树。我们要证明的是这两棵树的成本相同。如此也就证明了 Kruskal 算法可以找到最小成本生成树。令 $E(t)$和 $E(t')$分别为两棵树的连线集合（edge set），此时有两种可能：

(1) 两棵树的连线集合相同，则这两棵树的成本必相同，得证。

(2) 两棵树的连线集合不相同，则令 q 为落在 t 中(即$q \in t$)，但不在 t' 中的最小成本的一条连线。

① 此 q 必然存在，否则两棵树的连线集合相同。

② 将 q 加入 t' 中必产生唯一的环路，因为在任何一棵生成树（spanning tree）（图4.12粗线）上加入任意一条非树上的新连线（图4.12虚线）必产生唯一的环路（cycle）。

③ 在此唯一的环路中，必然至少存在一条连线不在 t 中，令 q' 为此连线（$q' \in t'$）。否则，此环路存在于此树 t 中（但 t 为一棵树，不应该有环路），产生矛盾。

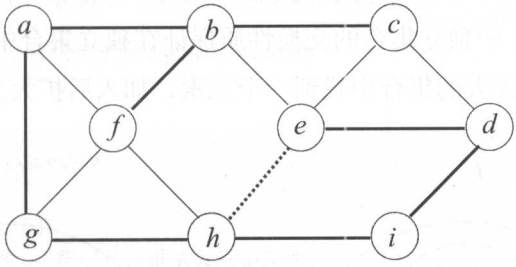


图4.12 产生唯一的环路

④ 连线 q'（$\in t'$）的成本必高于或等于连线 q（$\in t$）的成本。否则 q' 的成本小于连线 q 的成本，则 kruskal 的算法会先考虑将 q'（而非 q）纳入 t 中。注意，将 q' 纳入 t 中不会产生环路的原因是，小于 q 而落于 t 中的连线也必落于树 t' 中，因为当初选择的 q 是落在 t 中但不在 t' 中的一条最小成本线。

⑤ 将 t' 中的 q' 换成 q（即 $E(t')-\{q'\}\cup\{q\}$）会产生一棵新的生成树，且此树的成本不会高于原先的树 t'。也是一棵最小成本生成树，而其连线集合比 t' 更靠近 t。

重复以上步骤，t' 会逐渐转变成 t。因此得证。

看起来，想证明一个贪婪算法是正确的，有时不太容易。下面介绍一种离散结构——Matroid，可以协助证明一些贪婪算法。Matroid 的定义如下：

Matroid

Matroid 是符合下列条件的一个系统 $M=(S, I)$：

(1) S 是一个有限元素的集合（非空集合）。

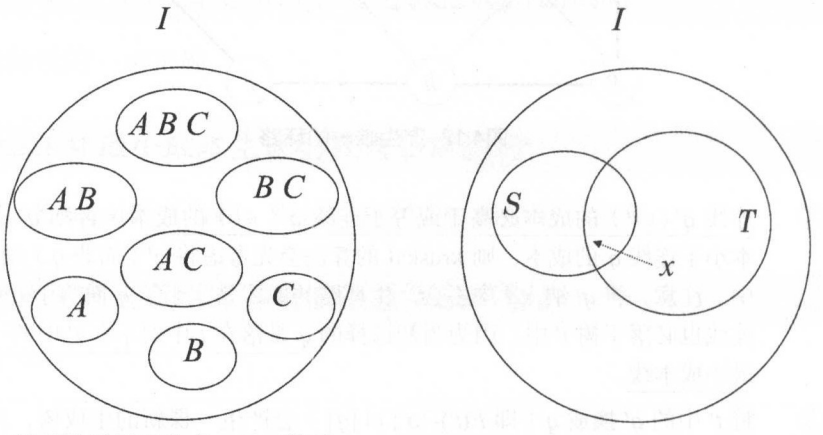
(2) I 是 S 的子集合所构成的集合（非空集合），称为 S的独立集合(independent set)，需具有以下两个性质。

① 继承性质(hereditary property): 如果 $B\in I$ 且$A\subseteq B$，那么 $A\in I$。

② 交换性质(exchange property): 如果 $A\in I$、$B\in I$，且 $|A|<|B|$，那么存在 $x\in B-A$ 使得 $A\cup\{x\}\in I$。

Matroid 中独立集合的继承性质是任何此集合中元素的子集合都落入独立集

合中。如图4.13所示，{*ABC*}落入独立集合中，则{*ABC*} 的子集合都落入独立集合中。Matroid 中独立集合的交换性质保证在独立集合中，任何一个较小的集合都可以从一个较大的集合中找到一个元素，加入后扩大为另一个独立集合。



继承性质需要集合{ABC}的子集合都落入独立集合中

交换性质存在 $x \in T\text{-}S$，使得 $S \cup \{x\} \in I$

图4.13

有些问题可表示成在有权重（正数）的 Matroid 中寻找最大权重的独立集合的问题。例如，最小成本生成树的问题就是寻找最小成本的连线，进而得以连接所有的顶点。将 {无回路的连线集合} 看成是一个独立集合，且将每条连线的权重*w转换成 c-w*，此处 *c* 是一个大于所有连线权重的正值，最小成本生成树的问题（可被证明）就是找出最大权重独立集合的 Matroid。

此问题的继承性质十分明显，因为无回路的连线集合的子集合必不含回路。而交换性质从表4.10的证明中大致可以看出。本章第4.5节中的单位时间工作调度问题也可以表示成 Matroid，其中的独立集合为 {不被罚钱的工作集合}。

当有一个问题可以被表示成有权重（正数）的 Matroid 时，表4.11中的贪婪算法可以找到最大权重的独立集合。

表 4.11 Matroid 的最优贪婪算法

输入	$M=(S, I)$：一个有权重的(weighted) Matroid
输出	A：最优解

步骤	Algorithm Greedy
	{
	Step 1: $A=\phi$。
	Step 2: 将 S 按照其权重从大到小排列。
	Step 3: 按照上述顺序，将 S 的元素 x 试着加入 A 中：If $A\cup\{x\}\in I$ (即独立集合)，then $A=A\cup\{x\}$。
	Step 4: 返回 A。
	}

4.7 贪婪法的技巧

贪婪法的一般设计步骤如下：

```
Algorithm Greedy(a, n)
// a[1:n] 包含 n 输入; solution 为解集合
{
solution:=ϕ;
for i:=1 to n do
  {
  按照某法则，从 a 中选择出 x。
  if 按照某法则判断 x 可以被加入 solution 中
  then
  solution:=solution∪{x}。
  }
返回solution。
}
```

当你想用贪婪法解题时，注意你的方法是否针对所有输入都可以找到想要的答案或最优解。若问题需要的答案是必须找出最优解，而你的方法只是找到还不错的解或偶尔找到最优解，则可能不符合要求。

另外，可表示成 Matroid 的问题可利用贪婪法解决或提供其正确性证明，但是不代表当一个问题不被表示成 Matroid 时，就断定此问题绝不可能使用贪婪法求解。

总而言之，使用贪婪法时要小心设计并证明，以免失去找到最优解的机会。最后，一般算法书中提到可以用贪婪法解的问题还有以下几个：

（1）磁带上的最优存储空间：有 n 个程序想被存储在一个长度为 L 的磁带上。假设每一个程序被读取时，磁带都在起始的位置。每个程序的长度可能不同，请找出n个程序的一种存储排列方式，使得其平均的读取时间最短。

（2）最优合并模式：两个排列好的稳健分别有s和t笔数据，共需要 O(s + t) 的时间，将其合并成一个排列好的文件。输入n个排列好的文件（大小不一），找出一个花费最少比较次数的合并模式，可以两两合并成最终的文件。

学习效果评测

1. 编写一个程序，将一个连通图 G = (V, E) 的最小成本生树的权重总和输出，其中 V 为顶点集合，而 E 为有权重的连线集合。

输入：

```
9                (顶点 V 的个数)
15               (连线 E 的个数)
a b 2            (以下是连线及其权重)
a f 13
a g 7
b c 6
b e 15
b f 9
c d 10
c e 12
d e 3
d i 7
e h 8
f g 11
f h 14
g h 5
h i 1
```

输出：
40　　(最小成本生树的权重总和)

2. 有n个程序想被存储在一个长度为 L 的磁带上。假设每一个程序被读取时磁带都在起始的位置。每个程序的长度可能不同，找出n个程序的一种存储排列方式，使其平均的读取时间最短。假设每一个程序被读取的概率是一样的。例如，当 n = 3，而3个程序的长度为 (5, 3, 1)时，不同的存储顺序将会有不同的读取时间。

输入：
3　　(程序的个数)
5 3 1　　(每个程序的长度)

输出：
1 3 5　　(最优的存储排列方式)

顺序	读取时间
5, 3, 1	5+(5+3)+(5+3+1)=22
5, 1, 3	5+(5+1)+(5+1+3)=20
3, 5, 1	3+(3+5)+(3+5+1)=20
3, 1, 5	3+(3+1)+(3+1+5)=16
1, 3, 5	1+(1+3)+(1+3+5)=14
1, 5, 3	1+(1+5)+(1+5+3)=16

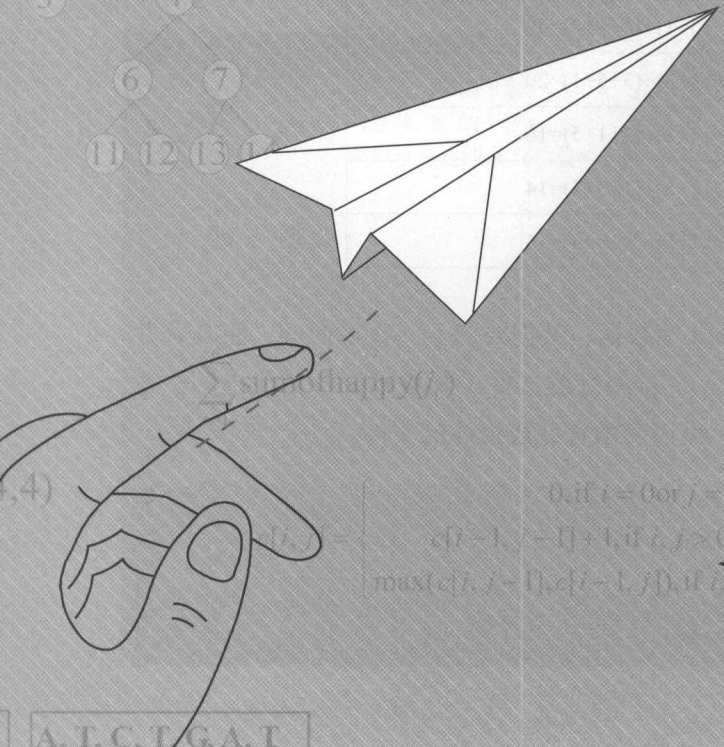
3. 两个排列好的文件分别有 s 和 t 笔数据，共需要 O (s + t) 的时间，将其合并成一个排列好的文件。输入 n 个排列好的文件（长度不一），找出一个花费最少比较次数的合并模式，可以两两合并成最终的文件。

输入：
20 30 10 5 30　　(每个文件的长度)

输出：
205　　(最少的比较次数)

修剪与搜索法

章节大纲

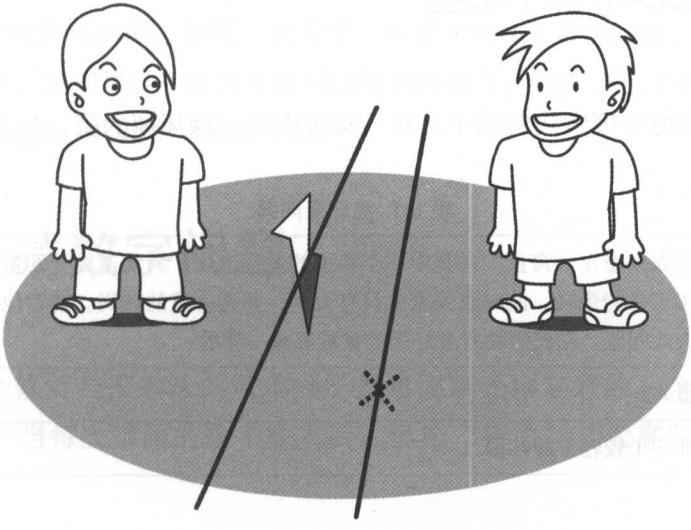


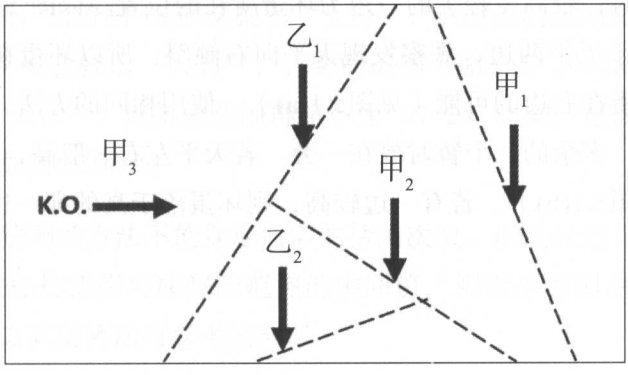
5.1　何谓修剪与搜索法

何谓修剪与搜索法（prune and search）？

简而言之，就是在寻找解的过程中，重复地将解的可能范围缩小，直到可以直接找到解为止。

修剪与搜索法好像儿时玩的游戏。





首先，用小刀在泥地上画一个长方形。第一个小朋友将此小刀向长方形中掷下，并使其插立于泥地上，并任意画一条直线经过此点，同时将此长方形分割成两半。接下来，另一个小朋友在此两半中选择其一掷下，并使其插立于泥地上；同样地，任意画一条直线经过此点，并将选择的区域分割成两半。

两方不断轮流，直到一方掷到选择的区域外或未插立于泥地上时宣告失败。这个游戏的特点是可以选择的区域不断变小，使得将小刀掷中的难度越来越高。此游戏和修剪与搜索法都是将求解空间不断缩小的一种方法。

5.2　找坏蛋问题

第一个修剪与搜索法的例子是从一窝蛋中找一颗坏掉的蛋，如表5.1所示。

表 5.1　找坏蛋问题

问题	有一位农夫想在一窝蛋中将其中一个坏了的蛋找出来。只知道坏了的蛋只有一个，而且比正常的蛋轻。此时农夫家中只有天平，并无称重的砝码。请替他设计一个算法解决此问题。在此假设所有好蛋的重量都是一样的。
输入	一窝蛋共 n 个
输出	找到唯一（较轻）的坏蛋

找到坏蛋的方法需要善用天平。因为坏蛋的重量较轻，所以当天平两边有相同数量的蛋时，较高（轻）的一边为坏蛋所在的位置。以图 5.1 为例，将一窝10个蛋平分于天平两边，观察发现天平向右倾斜，所以坏蛋在左边的5个蛋中，可排除坏蛋在右边的可能（见图5.1(a)）。使用相同的方法，再将5个蛋平分于天平两边，多余的一个暂时放在一旁。若天平左右一般高，则放在旁边的蛋是坏的（见图5.1(b)）。若有一边较高，则坏蛋落于高的那一边（见图5.1(c)和图 5.1(d)）。

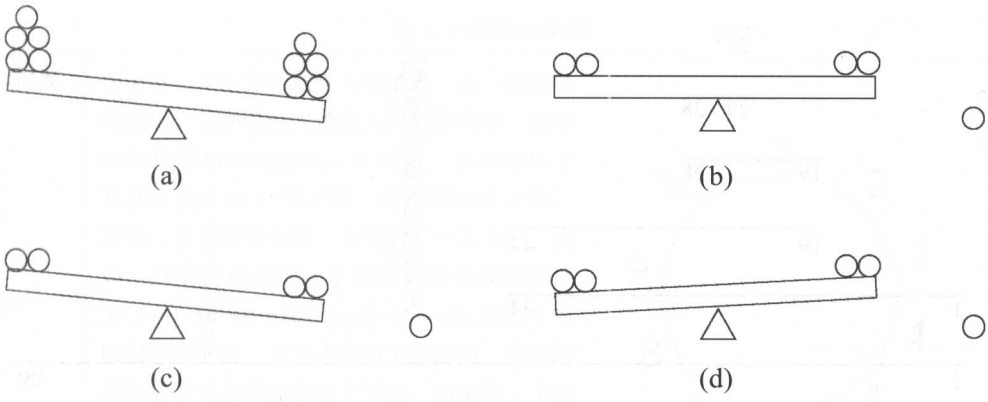


图5.1 坏蛋落于天平高的那一边

以上做法的特点是：每用一次天平，就将大约 1/2 的蛋排除。当剩下 2 个或 3 个蛋时，再用最后一次天平就可以找到坏蛋了。显然，这个算法的时间复杂度为$O(\log_2 n)$，此处 n 为蛋的个数。

5.3 猜数字问题

第二个修剪与搜索法的简单例子是猜测事先预定好的一个正整数（小于 100）。当每次猜测的数字输入后，会给“猜中”“太高”或“太低”的提示。

假设事先预定好的整数是 27，当猜测 45 时会提示“太高”，此时解的范围在 1~44 之间；当猜测 18 时会提示“太低”，此时解的范围缩小为 19~44 之间；当猜测 29 时会提示“太高”，此解的范围再缩小为 19~28 之间；当猜测 26 时会提示“太低”，此解的范围再缩小为 27~28 之间。显然，最多再猜两次即可猜中，如图5.2所示。

这个任意猜测的方法不能保证所需的猜测次数。但是经过几次尝试后可以知道，若每次总是猜测可能解的范围的中间值，则至少可消除一半的解的范围。这种猜测方式的算法可参考表5.2。

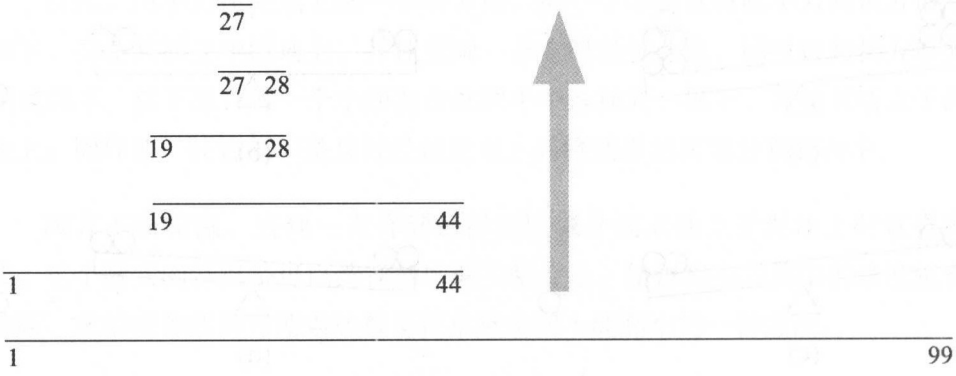


图5.2 解的范围不断地缩小

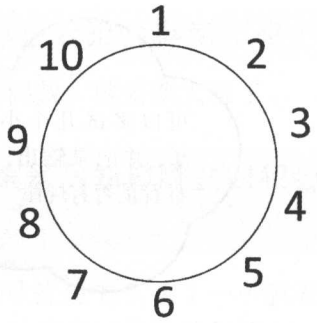
表 5.2 猜数字的算法

输入	1 到 100 之间的一个正整数 x
输出	“猜中”“太高”或“太低”的提示
步骤	Step 1：利用随机数函数产生 x，使得 $1 \leqslant x \leqslant 100$ Step 2：用户输入猜测的数字 z Step 3：如果$z=k$，那么输出“猜中”并停止此程序；如果 $z>x$，那么输出“太高”；如果 $z<k$，那么输出“太低” Step 4：跳至 Step 2

5.4 约瑟夫问题

接下来这个问题十分有趣，是一个利用数学拯救自己性命的一个历史故事，如表5.3所示。

表 5.3 约瑟夫问题

问题	西方有一个十分有数学天分的人，在一场战争战败后，他和其他人被敌人抓到监狱中。狱中的囚犯最后决定围成一个圆圈，并采取以下自裁行为：从 1 号开始，按照顺时针方向，每隔一个人就要自裁，直到剩下一人为止。例如，10名囚犯围成一个圆圈，则自裁的顺序为 2→4→6→8→10→3→7→1→ 9，最后5 号囚犯存活下来。这个人因为十分聪明，知道最后存活者是几号而存活了下来。请编写一个程序，输入囚犯人数 n，并输出存活下来的囚犯编号 $J(n)$。 
输入	囚犯人数 n 10 13
输出	最后存活下来的囚犯编号 $J(n)$ 5 11

哪一名囚犯会存活下来?

不知道呀!

可以试几个小例子，看看是否有规律?

当有 10 名囚犯时，自裁的顺序是 2→4→6→8→10→3→7→1→9，最后5号囚犯存活下来。

当有 13 名囚犯时，自裁的顺序是 2→4→6→8→10→12→1→5→9→13→7→3，最后11号囚犯存活下来。

是否有规律?

好像偶数编号的囚犯都最先自裁。

奇数是否有规律?

好像没有。

表5.4是具有1~16 名囚犯的16种情况下最后存活囚犯的编号。这里 J (n) 代表当有 n 名囚犯时，最后存活下来的囚犯编号。

表 5.4 约瑟夫问题的1~16名囚犯不同情况下的结果范例

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$J(n)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15	1

从此表中可以得知，当囚犯人数 J=1, 2, 4, 8, 16 时，其输出 $J(n)$ 都是 1。而且存活下来的囚犯编号都是奇数，因为偶数编号的囚犯在第一圈时，就最先自裁了。我们可先删除所有偶数编号的囚犯会存活到最后的可能。

哪一名奇数编号的囚犯会存活下来?

不知道呀!

在第一圈后，还存活几名奇数编号的囚犯?

大约一半的囚犯。准确地说，如果所有囚犯人数是偶数 $2k$，那么在

第一圈后，还存活 k 名奇数编号的囚犯；如果所有囚犯人数是奇数 $2k+1$，那么在第一圈后，还存活 $k+1$ 名奇数编号的囚犯。

知道还存活 k 名奇数编号的囚犯时，哪些囚犯会存活下来？

不知道呀！好像需要重新编号，一旦又被编到偶数，就要倒大霉了。

在第一圈偶数号囚犯自裁后，如果重新编号后知道哪些囚犯会存活下来，这些囚犯的旧编号是多少？

肯定是奇数！当新编号是 1, 2, 3, 4时，其旧编号就是 1, 3, 5, 7 ……就看新编号是第几名。如果新编号是第 i 名，那么对应的旧编号就是第 2i-1 名。这是一个等差数列。

可否用一个小例子检查上述发现是否为真？

假设一开始共有 10 名囚犯。在第一圈自裁 5 名偶数号囚犯后，剩下 5 名奇数号囚犯。在此剩下的 5 名奇数号囚犯中，谁会是最后的存活者？简单，查看表 5.4 就好了，$J(5)=3$。所以是顺时针排第 3 名的囚犯，他的原先编号是 1，3，5（$=2\times3-1$）。最后存活者是编号为 5 的囚犯。

如何利用符号记录上述发现？

$J(10)=5$。换句话说，在第一圈自裁 5 名偶数号囚犯后，$J(10)=2\times J(5)-1=2\times3-1=5$。

$J(2k)=2J(k)-1$？另一方面，$J(2k+1)=$？

嗯……

当囚犯人数是奇数 $2k+1$ 时，在第一圈删除编号 1 后，还存活 k 名奇数编号的囚犯，因此 $J(2k+1)=2J(k)+1$。若再加上先前的发现（即$J(2k)=2J(k)-1$），则可以设计一个算法解决约瑟夫问题，如表5.5所示。

表 5.5 约瑟夫问题的算法

输入	囚犯人数 n(>=1 的正整数)
输出	最后存活下来的囚犯编号 $J(n)$

步骤	Algorithm $J(n)$
	{
	if n=1 then 输出 $J(1)$=1。
	else {
	if n=2k, then $J(2k)$=2$J(k)$-1;
	else $J(2k+1)$=2$J(k)$+1;
	}
	}

公式 $J(2k)=2J(k)-1$ 也说明了“J=1, 2, 4, 8, 16 时，其输出 $J(n)$都是 1”的原因，如$J(4)=2J(2)-1=2(2J(1)-1)-1=1$。

其实还有一个更快、更直接的计算方式，这个方式巧妙地隐藏于二进制表示法中。表5.6 是将表5.4的数值化成二进制表示法，以方便进一步观察。

表 5.6 神奇的规律隐藏于二进制表示法中

n	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
$J(n)$	1	01	11	001	011	101	111	0001	0011	0101	0111	1001	1011	1101	1111

n 和 $J(n)$之间是否有规律?

好像出现 1 的数目相同。

排列顺序是否有规律?

好像没有。不，好像是在绕圈圈。

绕圈圈? 可以换个方式说清楚吗?

将 n 的每个数字向左移一格，同时最左的 1 像绕圈圈一样移到最右边，就变成 $J(n)$。

你是正确的，但是为什么呢？

嗯……

目前在什么情况下这个问题最容易被解决？

当 J=1, 2, 4, 8, 16 或 2^k 时，最后存活的囚犯编号必为 1。

可以利用这个结果吗，困难点在何处？

所有囚犯人数不会总是 2k，除非囚犯人数可以减少。

什么方法可以让囚犯人数减少？

如果可以，先让几名囚犯自裁。直到剩下 2^k 名囚犯，这样最后自裁的囚犯的下一位就是最后存活者。

需要自裁多少囚犯？

多于2^k 的囚犯就自裁掉。例如，J=10 时，需要自裁到剩下 2^3 名囚犯；也就是 $10=2^3+2$，需要自裁 2 名囚犯。

知道最后自裁的囚犯是谁就好了。

不难呀！因为 2 名囚犯要先自裁，所以自裁的顺序是从偶数的 2, 4, 6……算起；$2\times2=4$，最后自裁的是编号为 4 的囚犯。

那么最后存活的囚犯编号是多少？

最后自裁囚犯的下一位，就是编号为 4 的下一位；自然，4+1=5，编号为 5 的囚犯就是最后存活者了。

这个方法和先前观察的规律有关系吗？

当 $n=10=(1010)_2$ 时，将$(1010)_2$ 的每个数字向左移一格，得到$(0100)_2=4$，刚好是最后自裁的囚犯的编号。之后将最左的 1 移到最右边，刚好得到最后存活者的编号 $J(n)=(0101)_2=5$。真是神奇呀！

将10=$(1010)_2$ 的每个数字向左移一格，就得到最后自裁囚犯的编号，为什么？

因为$(1010)_2$-$(010)_2$=$(1000)_2$=8=2^3，$(010)_2$ 就是需要减少的囚犯的数目。将$(1010)_2$ 的每个数字向左移一格，$(010)_2$=2 乘以 2 变成$(0100)_2$，就是计算最后自裁囚犯的编号。

为何最左的 1 需要移到最右边？

最后自裁囚犯的下一位就是最后存活者，这个加 1 的操作只是在算下一名囚犯的编号。

表5.7 是利用此方法所设计出的快速算法。

表 5.7　约瑟夫问题的快速算法

输入	囚犯人数 n
输出	最后存活下来的囚犯编号 $J(n)$

步骤	Algorithm $J(n)$
	{
	Step 1: 将 n 表示成 2^k+m，使得 k 越大且 m 为正值。
	Step 2: 输出 $2m+1$。
	}

5.5　简化的线性规划问题

线性规划是一个应用广泛的解题工具，因此有效率地找到线性规划问题的解是一个程序设计的基本课题。首先，利用一个例子引出线性规划问题，再设计一个有效率的修剪与搜索算法解决线性规划中的一个简化问题。这是一位农夫想养猪羊赚钱的问题，可参考表5.8。

表 5.8　农夫养猪羊问题

问题	某一位农夫想在自己的农舍中养猪羊来赚钱。他向朋友借了 200 000 元当作创业资本。但是，一头猪或羊都需养 12 个月后才可以出售。购买一头小猪的成本需要 1 500 元，每个月饲养猪的成本是 400 元，一年后出售的价格是 22 000 元。相对地，购买一头小羊的成本需要 3 200 元，饲养羊的每个月成本是 700 元，一年后出售的价格是 35 000 元。请问这位农夫最少需要养几头猪和几头羊，才可以在一年后靠卖猪羊换得现金超过 200 000 元
输入	一头猪的购买成本、饲养成本及出售价格 1 500，400，22 000 一头羊的购买成本、饲养成本及出售价格 2 200，700，35 000
输出	养 x 头猪，y 头羊 0, 6

注：该范例中表示价格的数字只是举例用，没有实际市场意义

用 x 代表农夫饲养猪的数目，y 为饲养羊的数目。我们可以将上述农夫养猪羊的问题转换成以下两个变量的线性规划问题。

- 目标式（objective function）：极小化 x+y 的值（此式在于希望养猪羊的总数最少）。
- 限制式（constraints）：此解需符合下列每一个条件。

（1）$x \geq 0$ 且 $x \varepsilon Z$（即整数）。

（2）$y \geq 0$ 且 $y \varepsilon Z$（即整数）。

（3）1 500x+2 200y+400 × 12x+700 × 12y≤200 000（此式等同于6 300x +10 600y≤200 000，在于保证养猪羊的成本少于向朋友借的钱）。

（4）22 000x+35 000y≥200 000（此式在于保证卖猪羊的钱超过200 000 元）。

图解法常被用来解决简单的线性规划问题。此法先绘出其解空间，再寻找解空间的解，使其符合目标式的要求。上述问题对应的解空间如图5.3所示。

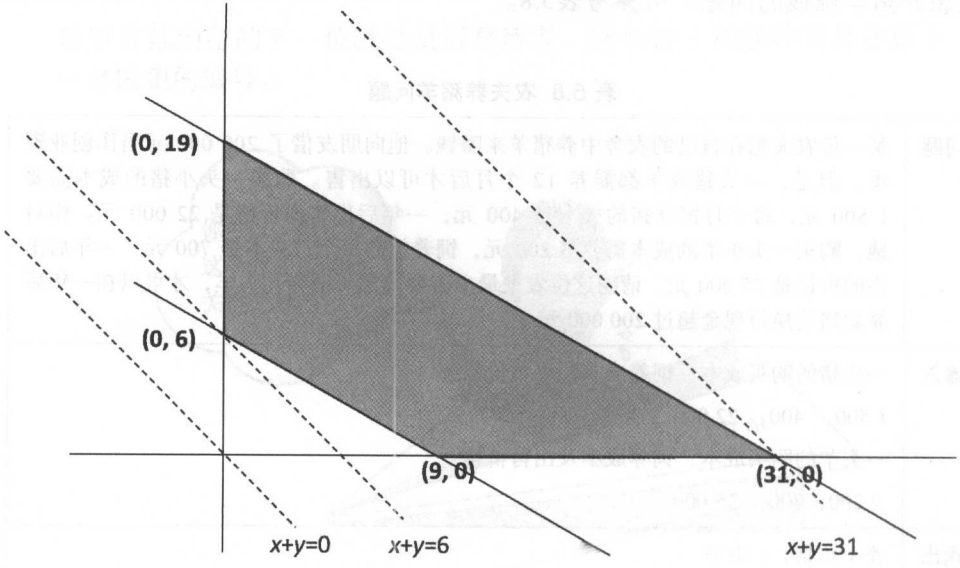


图5.3 对应的解空间

图 5.3 中任何落在解空间内（灰色区域）的点都符合限制式。表 5.9 的问题就在图 5.3 的灰色区域中，找到使得 $x+y$ 为最小的点。因此，图解法接下来寻找经过灰色区域，且和 $x+y=0$ 并行的线（即 $x+y=k$ 的直线），使得 k 值最小。从图 5.3 得知，$x+y=6$ 为此解。因此这位农夫需养 6 头羊，就可以在一年后靠卖羊获得超过 200 000 元。

此问题的最优解必定落在直线的交点上，因此图解法暗示着一个简单的算法。此法就是先计算出所有任意两条直线的交点，过滤去除不符合限制式的交点，再从中找到让目标式（objective function） x + y 为最小的交点，就是最优

解。当 n 代表限制式的数目时，因为 n 条直线最多有 $\binom{n}{2}=n(n-1)/2$ 个交点，所以这个算法最少需要 $O(n^2)$ 以上的时间执行。

设计更快的算法求解两个变量的线性规划问题看起来并不容易。我们将介绍一个简化后的线性规划问题，并且为其设计一个 $O(n)$ 时间复杂度的算法。有趣的是，这个算法在稍加修改后，有机会解决一般的两个变量的线性规划问题，其时间复杂度依旧是 $O(n)$。

此简化问题的目标式固定为 y，而其所有限制式都为 $y \geq ax+b$ 这种类型（这里的 a 和 b 是常数）。下面介绍一个简化的两个变量线性规划问题的范例，对应的解空间如图5.4所示。

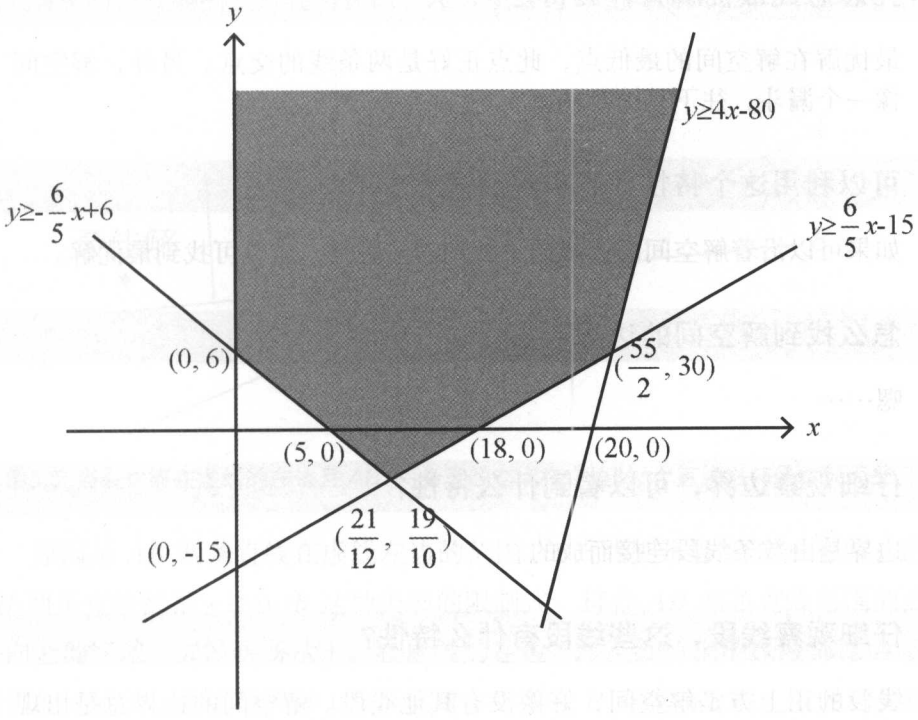


图5.4　对应的解空间

目标式：找出 y 的最小值
限制式：此解需符合下列条件

(1) $0 \geq -x$

(2) $y \geq -\frac{6}{5}x + 6$

(3) $y \geq \frac{6}{5}x - 15$

(4) $y \geq 4x - 80$

如何找到这个简化问题的最优解？

嗯……

先想想此最优解有什么特性？

最优解在解空间的最低点，此点正好是两条线的交点。另外，解空间像一个漏斗，往下是缩小的。

可以利用这个特性找到最优解吗？

如果可以沿着解空间的边界向下找到漏斗底部，应该可找到最优解。

怎么找到解空间的边界？

嗯……

仔细观察边界，可以看到什么特性？

边界是由数条线段连接而成的。

仔细观察线段，这些线段有什么特性？

线段的正上方是解空间，好像没有其他线段。解空间的边界总是出现在最高点。

试试看是否可利用这个特性找到边界？

利用一条自上而下的直线计算与其他直线所有的交点。形成最高交点

的线会帮助构成边界。

找到边界对寻找最优解有什么帮助？

最优解位于边界最低的方向，也就是计算边界的斜率可以决定最优解的方向。

如何利用最优解的方向协助找到最优解？

嗯……

利用最优解的方向修剪冗余的限制式，以便高效率地找到最优解，这是这一算法的核心。下面的范例说明其主要概念。在图5.5中，当最优解在虚线的左边且 *AB* 两条直线的交点在右边时，*A* 直线必与最优解无关。

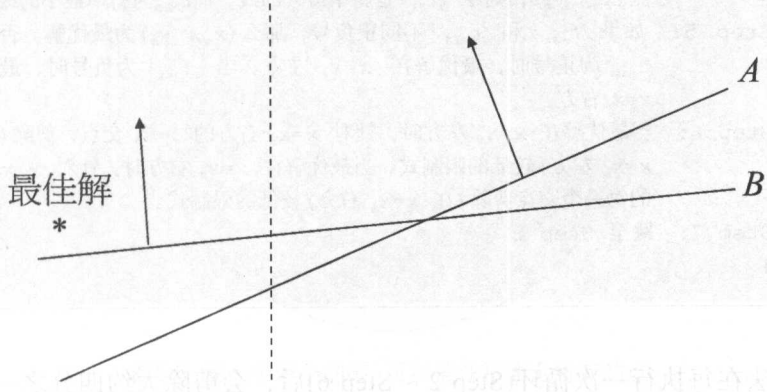


图5.5 当最优解在虚线的左边且 *AB* 两条直线交点在右边时，*A* 直线必与最优解无关

原因是 *AB* 两条直线在虚线左边的范围，*B* 直线则永远高于 *A* 直线；因为 *AB* 两条直线都是 $y \geq ax+b$ 这种类型的限制式，符合 *AB* 两条直线所围的点都是向上的区域（如箭头所示）。在虚线的左边，*A* 直线所围的区域都比 *B* 直线所围的区域大，故可行的解（含最优解）必落在 *B* 直线的上端。因此对找到最优解而言，*A* 直线是多余的，可被去掉。

值得注意的是，任意两条相交的直线必可删除其中一条，而不影响找到最优解的机会。

表5.9的算法就是利用修剪搜索的概念不断修剪去掉一定比例的限制式，直到剩下两条限制式为止。

表 5.9 简化的两个变量线性规划问题的算法

输入	目标式：找出 y 的最小值
	限制式：$y \geq a_i x_i + b_i$ $(i=1...n)$
	同样斜率的限制式，只保留最高的一条限制式
输出	符合限制式的最优（小）解

步骤	Algorithm simplified_linear_programming
	{
	Step 1: 如果未超过两条限制式，那么直接解开。
	Step 2: 将限制式两两任意配对，并解开其交点。用x_j代表这些交点的x坐标。
	Step 3: 找出 x_j 的中间值并令其为 x_m。
	Step 4: 找出与 $x=x_m$ 产生交点最高 $(y=y_m)$ 的直线。若有两条直线以上交于此高点 (x_m, y_m)，则令 t_{max} 是斜率最大的线，而t_{min}为斜率最小的线。
	Step 5: 如果 t_{max} 和 t_{min} 不同正负号，那么 (x_m, y_m) 为最优解。否则，当 t_{max} 为正号时，最优解在 $x=x_m$ 左方。当 t_{max} 为负号时，最优解在 $x=x_m$右方。
	Step 6: 当最优解在 $x=x_m$ 左方时，针对 $x=x_m$ 右方的每一个交点，剪除（在 $x=x_m$ 左方）较低的限制式。当最优解在 $x=x_m$ 右方时，针对 $x=x_m$ 左方的每一个交点剪除（在 $x=x_m$ 右方）较低的限制式。
	Step 7: 跳至 Step 1。
	}

此算法在每执行一次循环(Step 2 ~ Step 6)后，会剪除大约四分之一的限制式。因为当有 n 个限制式时，在 Step 2 中会找到大约 $n/2$ 个交点，而利用直线 $x=x_m$ 分割成左右各大约 $n/4$ 个交点后，最后将会剪除大约 $n/4$ 个限制式。

令 $T(n)$ 是执行此算法所需的时间时，可得 $T(n)=T(3n/4)+O(n)$，因为 Step 1 到 Step 7 都可以在 $O(n)$ 时间内完成（注意，Step 3 可利用第2章找第 k 小值的 $O(n)$ 算法）。解开此递归式可知，此算法的时间复杂度为$T(n)=O(n)$。

5.6　修剪与搜索法的技巧

修剪与搜索法总是需要执行多次循环，并在每次循环中缩小固定比例的解范围。如何有效地缩小固定比例的解范围是使用修剪与搜索法的关键技巧。

最后列出一个可被修剪与搜索法解决的问题——一个中心问题（the 1-center problem）：给定平面上 n 个点，找一个最小的圆可以覆盖所有的点，如图5.6所示。

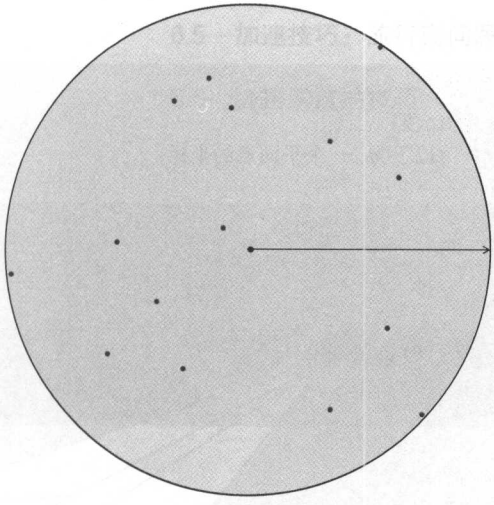


图5.6 找一个最小的圆，可以盖住平面上所有的点

学习效果评测

1. 约瑟夫问题：编写一个程序，输入囚犯人数 n，输出存活下来的囚犯编号 $J(n)$。

```
输入
10                    (囚犯人数 n)

输出
5                     (存活下来的囚犯编号 J(n))
```

2. 假设有 $2n$ 个人围成一个圆圈。前面 n 个人是好人，而后面 n 个人是坏人。编写一个程序，找出一个 m 值，当沿着圆圈处决第 m 人时，所有坏人会先被定罪。

```
输入
3                    (n值)

输出
5                    (m值)
```

3. 输入平面上 n 个点，编写一个程序，找出一个最小的圆（circle）可以包含平面上所有的点。

```
输入
 4                   (n值)
 1   0               (以下为 n 个平面点的坐标)
 0   1
-1   0
 0  -1

输出
0 0                  (圆心的坐标)
1                    (圆的半径)
```

树搜索法

章节大纲

6.1　何谓树搜索法

什么是树搜索法(tree searching)?

简而言之，就是将问题的解空间想象成一棵树，求解的过程如同在 这棵树上搜寻答案。

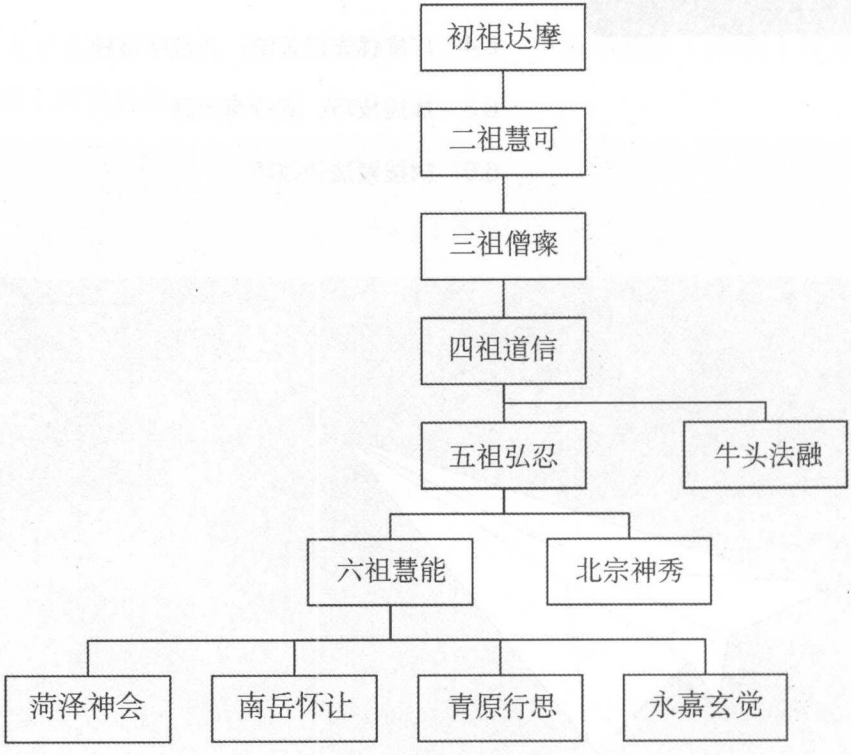


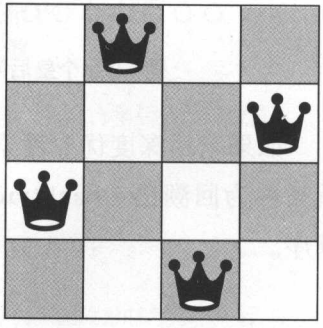
图6.1　禅宗法脉传承

许多问题的解空间可以表示成一棵树。例如，禅宗法脉传承（见图6.1）、生物的演化、人类家族的繁衍都可以利用树来展现。在树上搜索解的过程如同在族谱上寻找自己的亲人一般自然。

6.2　树状解空间：n 个皇后问题

第一个例子是 n 个皇后问题，如表6.1所示。

表 6.1　n个皇后问题

问题	将 n 个皇后放在一个 $n \times n$ 的棋盘上，使得彼此不会相互攻击。也就是没有两个皇后被放在同一行、同一列或同一个对角线上
输入	皇后的个数 n n=4
输出	输出 n 个皇后放在 $n \times n$ 的棋盘上的位置，使得她们彼此不会相互攻击 右图是4个皇后可能放置的位置(♛ 代表皇后) 

若将所有皇后按照行的顺序编号，则此处的 X_i 代表第 i 个皇后，放在该行从左开始计算的位置(即 $1 \leqslant i \leqslant 4$ 且 $1 \leqslant X_i \leqslant 4$)。例如，4个皇后问题其中一个解可以表示成$(X_1, X_2, X_3, X_4)$=(2, 4, 1, 3)（可参考表6.1中的输出）。

树搜索法的第一步需将问题的解空间想象成一棵树。图6.2以4个皇后的问题为范例。

这棵树的第一层（level 1）代表第一个皇后所放的行位置 X_1；第二层（level 2）代表第二个皇后所放的行位置 X_2；类似地，第 i 层（level i）代表第 i 个皇后所放的行位置 X_i。任意从这棵树的树根（root）走到树叶（leaf）的一条路径就代表4个皇后可能放置的位置。因为同一行不能放置两个皇后，即从树根走到树叶的任意一条路径上的值不会重复出现。

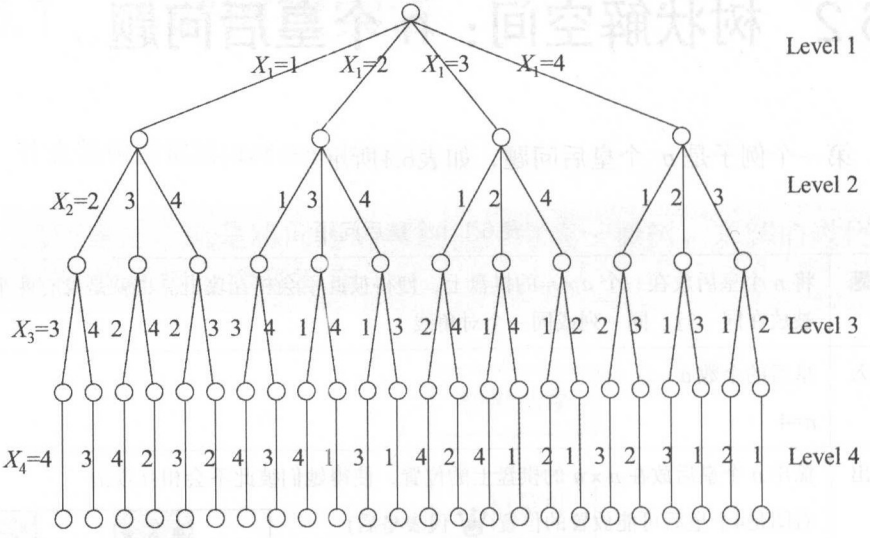


图6.2 n个皇后问题的解空间被想象成一棵树（当 n = 4时）

如果利用深度优先搜索法（depth-first search）的方式在这棵树上搜索，就被称为回溯法（backtracking）。如图6.3所示，树上的数字代表此搜索的顺序。

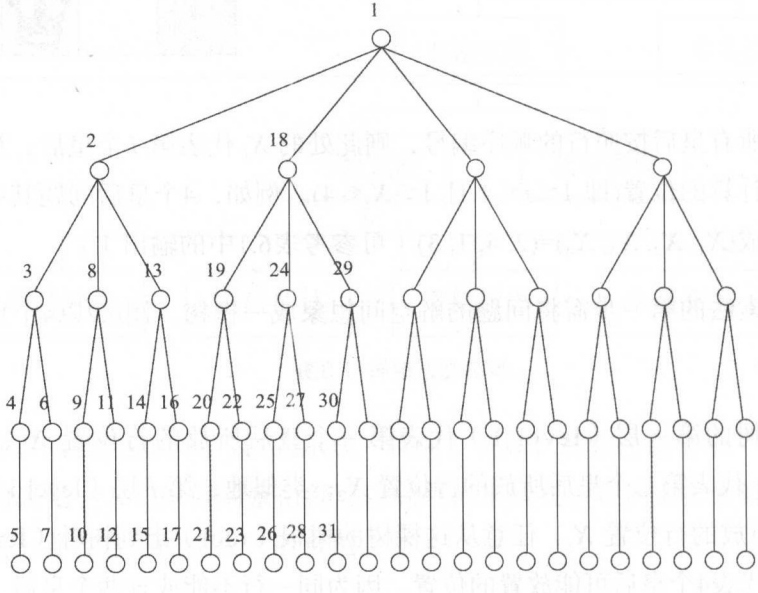


图6.3 在 4 个皇后问题的树状解空间上，利用回溯法搜索的前 31 个步骤

在图6.3上，回溯法从编号 1 节点开始，按照数字的大小顺序搜索解答。当遍历第 31 个节点时，找到4个皇后的第一个解。此时，从树根走到树叶的一条路径为 1-18-29-30-31，其对应到图6.2的解为(X_1, X_2, X_3, X_4)=(2, 4, 1, 3)。此解正好是表6.1中4个皇后的放置位置。

表6.2将介绍4个皇后的撤退法。

表 6.2　4个皇后的撤退法

输入	无
输出	X(1:4)

步骤	Algorithm four_queen_backtrack
	{
	integer k, X(1:4) //k 代表当前考虑的皇后的编号
	//X(1:4)存储4个皇后的行数
	
	X(1)←0; k←1 //设置当前考虑的皇后为第 1(k=1)皇后，其行位置为 0(X(1)=0)
	while k>0 do
	{
	X(k)←X(k)+1 //将第 k 个皇后的位置移到下一行
	
	while X(k)≤4 而且不可安全地放置此皇后时 //当不能放置时考虑下一行
	do X(k)←X(k)+1 //直到找到或 X(k)=5 为止
	
	if X(k)≤4 then //当目前考虑的皇后找到位置时
	if k=4 then 输出 X(1:4) //当找到全部4个皇后的解时，输出其位置
	else {k←k+1; X(k)←0} //否则寻找下一个皇后的位置
	else k←k-1 //当前考虑的皇后无法在此列找到合适的位置时
	//回溯回去，即继续考虑上一个皇后下一行的位置
	}
	}

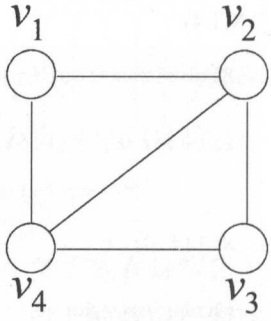
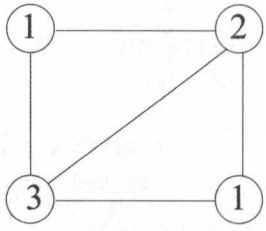
注意，上述算法并未构造出一棵树，在这棵树上搜索解答，也就是树搜索法用树代表解空间可能只是想象中的数据结构。

一般而言，一个问题的解空间可能使用不同种树表示。当然，就算在同一棵树上，也可以有不同的搜索顺序。下面将介绍常用的搜索方式。

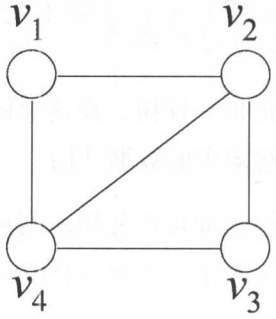
6.3　回溯法：涂色问题

第二个例子是涂色问题，也是一个可以使用回溯法解题的例子，如表6.3所示。

表 6.3 涂色问题

问题	这是一个涂颜色的游戏。游戏的内容是使用 k 种颜色将一个图的顶点（vertex）涂上颜色，使得相邻两个顶点的颜色不同
输入	一个图 G 和颜色数 k k=3 
输出	利用 k 种颜色将输入图的顶点涂上颜色，使得相邻两个顶点的颜色不同。 以下是当 k=3时的一种涂色 

同样地，首先将涂色问题的解空间想象成一棵树。图6.4是以表6.3中的输入为范例。此处的 X_i 代表点 v_i 的颜色编号（即 $1 \leqslant i \leqslant 4$ 且 $1 \leqslant X_i \leqslant 3$）。



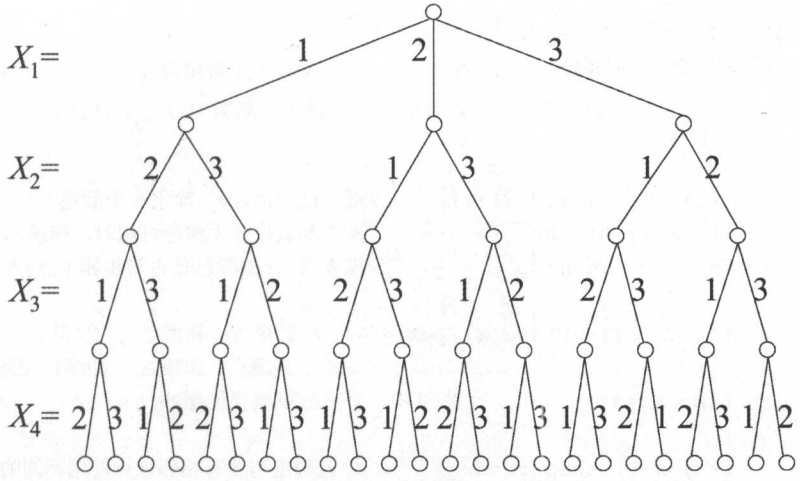


图6.4　4点图(使用三种颜色)的涂色问题其解空间可以想象成一棵树

同样地，在这棵树上进行深度优先搜索所得到的解为 (X_1, X_2, X_3, X_4)=(1, 2, 1, 3)。涂色问题的回溯法如表6.4所示。

表 6.4　涂色问题的回溯法

输入	一个图 G 和颜色数 k
输出	利用 k 种颜色将输入图上的顶点涂上颜色，使得相邻两个顶点的颜色不同

步骤	Algorithm Graph_Coloring(i)
	{
	//X[i]存储顶点 v_i 的颜色(1≤i≤n)，起始值都为 0 (代表未被涂上颜色)
	//参数 i 是下一个被涂色的顶点的编号
	//算法从执行 Graph_Coloring[1]开始
	{
	Repeat
	{
	调用 Next_Value(i);　//调用 Next_Value(i)获取 X[i] 的下一个颜色
	if (X[i]=0) then return; //无颜色可用于点 v_i 时，则回溯回到上一个顶v_{i-1}
	if(i=n)then write(X[1:n]);//点都被涂上颜色，则输出各顶点的颜色
	else Graph_Coloring(i+1);　//否则调用 Graph_Coloring(i+1)涂下一个顶点 v_{i+1}
	} until (false);
	}

步骤

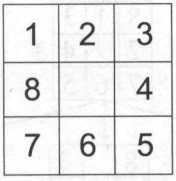
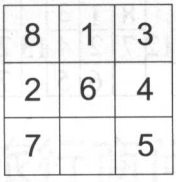
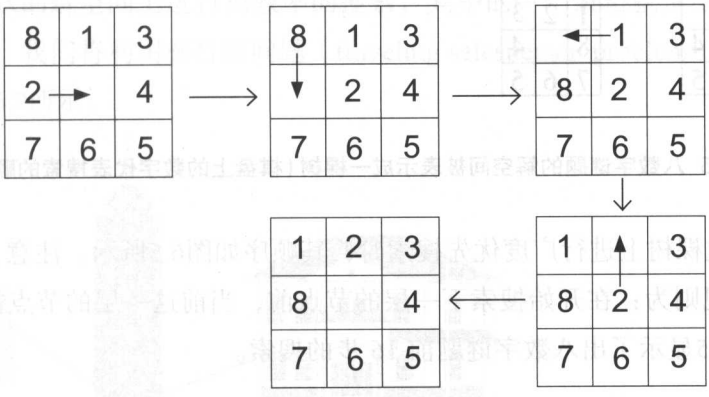
```
Algorithm Next_Value(i)
//返回顶点 i 的颜色
{
    Repeat
    {
    X[i]=X[i]+1 mod (k+1);    //试一试为顶点v_i 涂下一个颜色
    if (X[i]=0) then return; //若全都试过并无颜色可用时，则跳回
    for j:=1 to n do          //检查顶点i的颜色是否与相邻顶点j相同
    {
    if((G[i,j]<>0)and(X[i]=X[j])) //若顶点v_i和顶点v_j 是邻居
                                  且顶点v_i和顶点v_j 的颜色相同
    then break;                   //则跳出此循环
    }
    if(j=n+1) then return;        //若确定与所有相邻顶点使用不同的颜色
                                  //则跳回
    } until (false);
}
```

注意，在涂色问题的回溯法中，若无法顺利为顶点 v_i 涂上颜色（即所有颜色都与其相邻顶点冲突时），则回溯到上一个顶点v_{i-1}，并继续尝试为顶点 v_{i-1} 涂下一个颜色。此算法也并未构造出一棵树。

6.4 广度优先搜索法：八数字谜题

回溯法是在一个问题的解空间（表示成一棵树）上进行广度优先搜索。当然，在解空间中也可采取不同的搜索策略。下面介绍利用广度优先搜索法（breadth-first search）解八数字谜题，首先介绍八数字谜题，如表6.5所示。

表 6.5 八数字谜题

问题	一个 3×3 的棋盘上，任意放置 1~8 的数字。其中有一个空格未放数字。每次移动一个数字，使得最终的数字排列如下： 
输入	一个 3×3 的棋盘上，任意放置 1~8 的数字 
输出	每次移动一个数字，使得最后排出最终的数字排列 

这个问题的解空间可表达成图 6.5所示的一棵树。

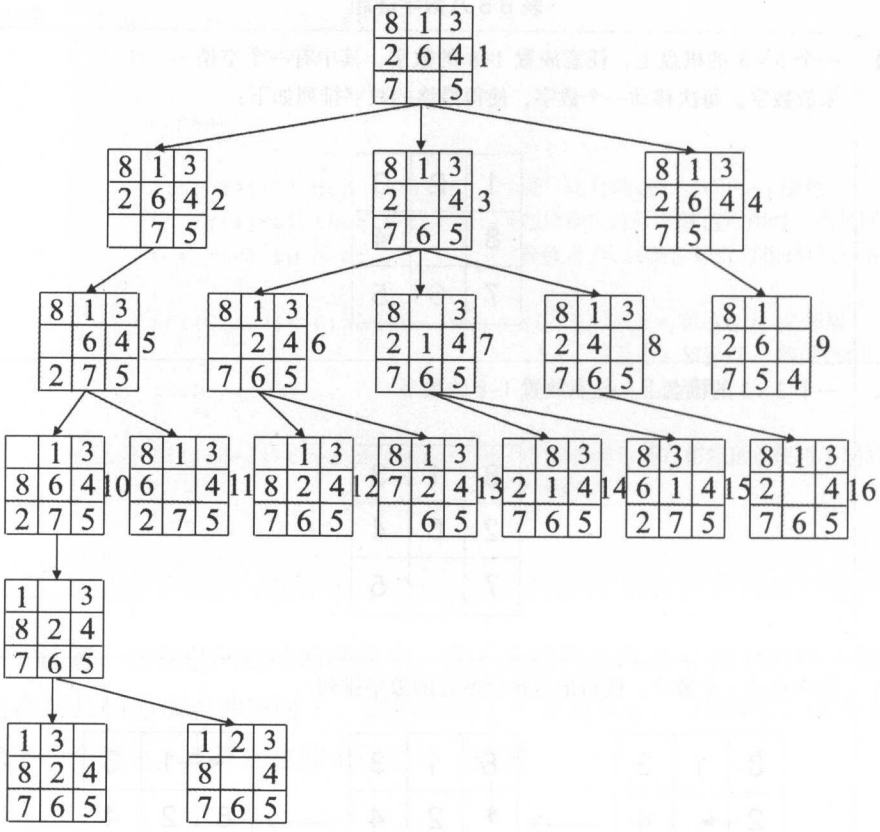


图6.5 八数字谜题的解空间被表示成一棵树(棋盘上的数字代表搜索的顺序)

当在这棵树上进行广度优先搜索时，其顺序如图6.5所示。注意，广度优先搜索法的规则为：在开始搜索下一层的节点前，当前这一层的节点需先被搜索完毕。图6.5显示了出八数字谜题前 16 步的搜索。

利用广度优先搜索法解八数字谜题的方法如表6.6所示。

表 6.6 八数字谜题的广度优先搜索法

输入	一个 3×3 的棋盘上，任意放置 1~8 的数字
输出	每次移动一个数字，使得最后排出最终的格盘

步骤	Algorithm Eight_Puzzle
	//每个节点代表棋盘的一个格子，将解空间想象成一棵树。在此树中，父格子可移动一个数字以转换成子格子，但此数字必须回避上次移动的数字，以加速寻找
	{
	Step 1: 将输入的格子加入 queue 中。
	Step 2: 检查在 queue 中的第一份数据是否为最终棋盘布局。如果是，就停止。
	Step 3: 剪除 queue 中的第一份数据，并将此数据的儿子加入queue的尾部。
	Step 4: 如果 queue 是空的，就输出寻找失败的信息，并停止；否则执行 Step 2。
	}

6.5 加速技巧：旅行商问题

若想在树状的解空间上进行高效率的搜索，其中的一个策略就是尽量省略不必要的搜索。我们将利用旅行商问题（traveling salesperson problem）说明这个技巧，如表6.7所示。

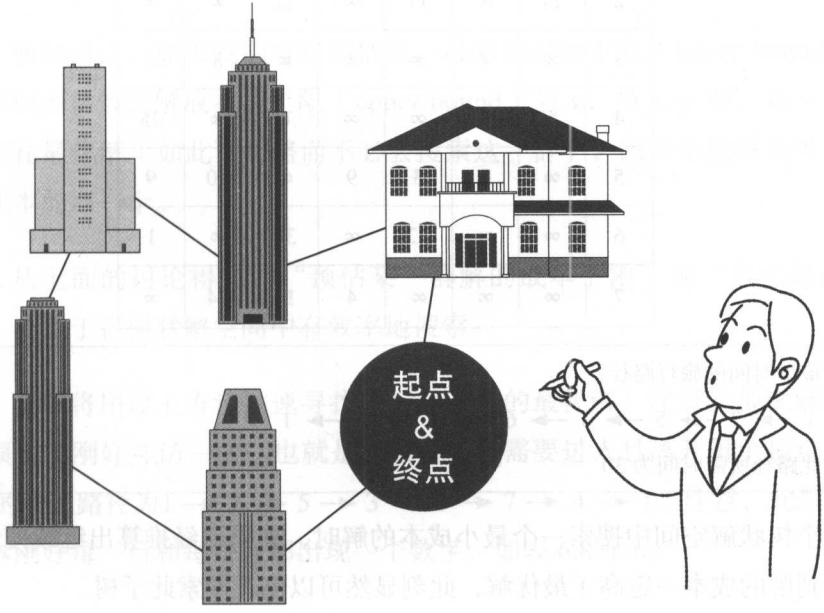
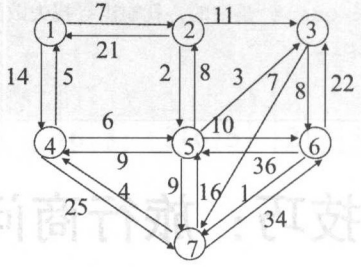


表 6.7　旅行商问题

<table>
<tr><td>问题</td><td>有一位超级推销员拥有许多客户，而客户分散于各地。在年终的时候，这位推销员想要送礼给每一位客户。在得知客户之间所需的交通时间后，帮这位推销员找到一个花费最少时间的旅行路径，使得每一个客户刚好被拜访一次，且最后需回到原出发点</td></tr>
<tr><td>输入</td><td>矩阵 $A[I, J]$存储从客户I直接到客户J所需要的交通时间。矩阵A也可以被表示为一个有向图G。此图的两顶点 I、J 代表两位客户的位置，而有向连线$[I, J]$上面的成本（非负数）代表由客户I到客户J所需要的交通时间（若客户I不能直接到客户J，则 $A[I, J]=\infty$）
 

$A[7, 7]$
<table><tr><td></td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr><tr><td>1</td><td>∞</td><td>7</td><td>∞</td><td>14</td><td>∞</td><td>∞</td><td>∞</td></tr><tr><td>2</td><td>21</td><td>∞</td><td>11</td><td>∞</td><td>2</td><td>∞</td><td>∞</td></tr><tr><td>3</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>∞</td><td>8</td><td>7</td></tr><tr><td>4</td><td>5</td><td>∞</td><td>∞</td><td>∞</td><td>6</td><td>∞</td><td>25</td></tr><tr><td>5</td><td>∞</td><td>8</td><td>3</td><td>9</td><td>∞</td><td>10</td><td>9</td></tr><tr><td>6</td><td>∞</td><td>∞</td><td>22</td><td>∞</td><td>36</td><td>∞</td><td>1</td></tr><tr><td>7</td><td>∞</td><td>∞</td><td>∞</td><td>4</td><td>16</td><td>34</td><td>∞</td></tr></table></td></tr>
<tr><td>输出</td><td>最少时间的旅行路径：
1 → 2 → 5 → 3 → 6 → 7 → 4 → 1
此路径所需时间为 30</td></tr>
</table>

在一个树状解空间中搜索一个最小成本的解时，如果已经推算出继续搜索某子树所得到解的成本一定高于最优解，此刻显然可以停止搜索此子树。

但是尚未找到最优解前，怎么知道某些解的成本高于最优解的成本呢？

好像是呀！

一定要知道最优解的值后，才能判断一个可行解的成本高于最优解的成本吗？

不知道！

譬如要找出全班同学中最矮的人（假设是 a），在我们不知道 a 是谁之前，如何判断 b 不是最矮的人？

只要 b 比任何一个人高，就一定不是最矮的人。

又譬如全班同学排成几排，如果甲排最矮的都比乙排最高的人都还要高，那么哪一排的人不会有最矮的人？

当然是甲排中不会有最矮的人。因为甲排最矮的人都比乙排最高的人高。

换句话说，如果我们可以预估某一群解的成本下限（lower bound）为 x，也可以预估最优解成本的上限（upper bound）为 y，当 $x>y$ 时，这一群解中必不存在最优解（如此可省略而不必去搜索这一群），因为最优解是可行解中最小成本的那一个。

从上面的讨论得知，“预估某一群解的成本下限”和“最优解成本的上限”有助于在树状解空间中有效率地搜索。

我们将用以上方法加速寻找旅行商问题的最优解。注意，最优解必须将每个顶点都刚好拜访一次，也就是此旅行路径需要进入且离开每个顶点一次。表6.7的最优路径为1 → 2 → 5 → 3 → 6 → 7 → 4 → 1。注意，此最优路径的成本刚好每一行和每一列都出现一个数字，如表 6.8所示。

表 6.8 在有向图矩阵 A 中，最优路径的成本刚好每一行和每一列都出现一个数字

	1	2	3	4	5	6	7
1	∞	⑦	∞	14	∞	∞	∞
2	21	∞	11	∞	②	∞	∞
3	∞	∞	∞	∞	∞	⑧	7
4	⑤	∞	∞	∞	6	∞	25
5	∞	8	③	9	∞	10	9
6	∞	∞	22	∞	36	∞	①
7	∞	∞	∞	④	16	34	∞

如果我们将任意一行或任意一列都同样减去一个常数（使其剩余值不为负值），那么此最优旅行路径并没有任何改变。例如，将表6.8中的第一行都减去一个常数 7，此举导致顶点 1 直接走到其他顶点的成本都同时下降 7。因为路径的其他部分成本都不变，故每一条旅游路径的相对成本差还是一样的，如表6.9所示。所以新的有向图的最优路径还是原来的那一条(1 → 2 → 5 → 3 → 6 → 7 → 4 → 1)。注意，这个被减去的常数是所有路径在这一步最少需要付出的成本。例如，从表6.9中得知，从顶点 1 直接走到其他顶点最少需付出的成本为 7。

根据同样的道理，只要任意一行（或任意一列）减去某一个常数，而不会出现负数，上述动作就可以重复执行，而最优路径依旧存在于最后的那个有向图中。将所有减去的常数进行加总，便成为所有路径成本的一个下限。

表 6.9 第一行都减去一个常数 7 后的有向图矩阵A

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	21	∞	11	∞	2	∞	∞
3	∞	∞	∞	∞	∞	8	7
4	5	∞	∞	∞	6	∞	25
5	∞	8	3	9	∞	10	9
6	∞	∞	22	∞	36	∞	1
7	∞	∞	∞	4	16	34	∞

每一行都减去一个最大的常数，使得该行不会出现负数，最后的结果如表6.10所示。当前累积的成本为 7+2+7+5+3+1+4=29。

表 6.10 第 1 行都减少 7，第 2 行都减少 2，第 3 行都减少 7，第 4 行都减少 5，第 5 行都减少 3，第 6 行都减少 1，第 7 行都减少 4 后的有向图矩阵 A

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	0	∞	∞
3	∞	∞	∞	∞	∞	1	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	7	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	30	∞

检查表6.10矩阵 A 的每一列，发现第 6 列并没有 0 。因此，第 6 列可以都减去 1，此旅行商问题所有解的一个下限是 29+1 = 30。经调整后的新有向图矩阵如表6.11所示。

表 6.11 将表 6.10 的第 6 列都减去 1 后的有向图矩阵A

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	0	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	29	∞

下面我们将思考如何将旅行商问题的解空间表示成一棵树。

假设此旅行路径选择经过 [2, 5] 这条线，那么此路径的最小成本是 30。因为此类的旅行路径都经过[2, 5]，每一个顶点仅能通过一次，故此类旅行路径不会在剩余的旅程中，而且不会从顶点 2 出发到其他顶点，或从其他顶点抵达顶点 5。我们可剪除第2行和第5列，而且 [5, 2] 这条线也不会出现在此路径中（否则会造成回路），因此可将 A[5, 2]设置为 ∞。经调整后的新有向图矩阵如表 6.12 所示。

表 6.12 所有路径经过 [2, 5] 的有向图矩阵 A(删除第 2 列和第 5 列后将A[5, 2]设置为∞)

	1	2	3	4	6	7
1	∞	0	∞	7	∞	∞
3	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	∞	20
5	∞	∞	0	6	6	6
6	∞	∞	21	∞	∞	0
7	∞	∞	∞	0	29	∞

相反地，当旅行路径不经过[2, 5]时，可将A[2, 5]设置为∞，如表 6.13所示。

表 6.13 将 A[2, 5]设置为∞后的方向图矩阵 A

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	19	∞	9	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	1	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	35	∞	0
7	∞	∞	∞	0	12	29	∞

虽然这些路径不经过 [2, 5]，但所有旅行路径又必须经过 2、5 这两个顶点。所以此路径一定从顶点 2 直接走到顶点 5 以外的顶点（即顶点 1 或顶点 3），而且从顶点 2 以外的顶点（即顶点 4、顶点 6 或顶点 7）直接走进顶点 5。从顶点 2 直接走到顶点 1 或顶点 3 的最小成本为 9（即 19 和 9 中的小值），而从顶点 4、顶点 6 或顶点 7 直接走进顶点 5 的最小成本为 1（即 1、35 和 12 中的最小值）。因此，不经过 [2, 5] 这条线的所有路径的成本从 30 增加了 10（= 9+1 ），即为 40。

同样地，第2行都减少9且第5列都减少 1，经调整后的新有向图矩阵如表6.14所示。

表 6.14 所有路径不经过 [2, 5] 的有向图矩阵*A*

	1	2	3	4	5	6	7
1	∞	0	∞	7	∞	∞	∞
2	10	∞	0	∞	∞	∞	∞
3	∞	∞	∞	∞	∞	0	0
4	0	∞	∞	∞	0	∞	20
5	∞	5	0	6	∞	6	6
6	∞	∞	21	∞	34	∞	0
7	∞	∞	∞	0	11	29	∞

使用 [2, 5] 将解空间分成两个子树，并且找到相对应的成本下限，如图6.6所示。

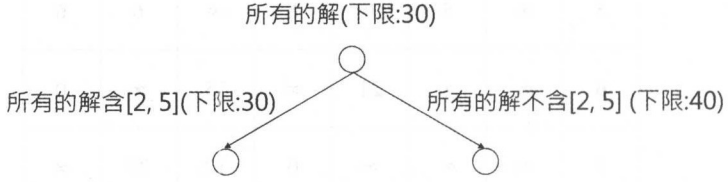


图6.6 使用 [2, 5] 分割解空间

以上做法可以持续进行，最后可以得到如图6.7所示的树状解空间。

从树根（root）到最左下的树叶（leaf）所形成的路径包含 [2, 5]、[7, 4]、[3, 7]、[6, 3]、[1, 2]、[4, 1]、[5, 6]，刚好可形成一个旅游路径 1 → 2 → 5 → 6 → 3 → 7 → 4 → 1，其成本为 57，可为当前“最优解成本的上限”。

在搜索整个树状解空间时，当某一子树“预估的成本下限”大于此“最优解成本的上限”时，此子树就可以省略而不必继续搜索。

例如，图6.7中含 [2, 5] 但不含 [7, 4] 的所有解（被打 × 的节点）的下限为 65，已经超过 57（当前最优解成本的上限），因此不必继续搜索。

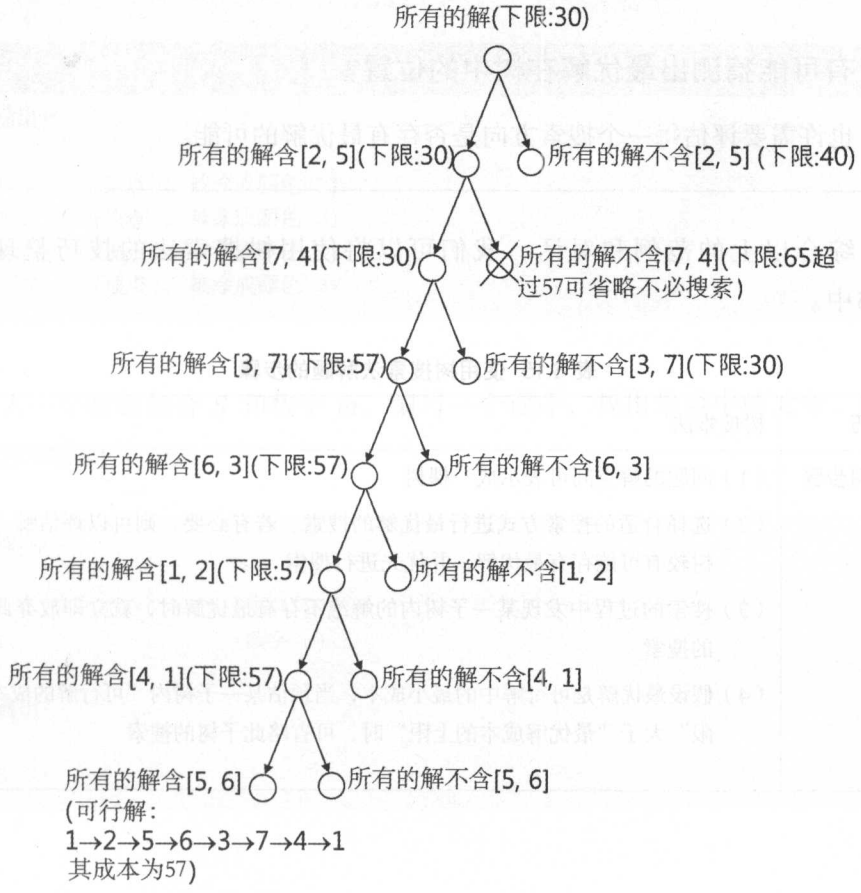


图6.7　当搜索到被打 × 的节点时，因为其“预估的成本下限”大于当前“最优解成本的上限”，因此不必继续搜索此子树

寻找旅行商问题最优解的过程中，发现在一棵子树下一定找不到可行解时（如已经被选择的路线构成回路，但尚有其他顶点未被拜访），也可以立刻停止这棵子树的搜索。

6.6 树搜索法的技巧

回溯法和广度优先搜索法哪一种会最快找到答案？

很难说呀！也许要看最优解在树中的位置。

有可能猜测出最优解在树中的位置？

也许需要评估每一个搜索方向是否存有最优解的可能。

综合以上的范例和对话，我们可以将使用树搜索法的技巧整理在表6.15中。

表 6.15 使用树搜索法解题的步骤

技巧	树搜索法
使用步骤	（1）问题的解空间可表示成一棵树 （2）选择合适的搜索方式进行最优解的搜索。若有必要，则可以评估哪一棵子树较有可能存有最优解，并优先进行搜索 （3）搜索的过程中发现某一子树内的解绝不存有最优解时，就立即放弃此子树的搜索 （4）假设最优解是可行解中的最小成本，当预估某一子树内“可行解的成本下限”大于“最优解成本的上限”时，可省略此子树的搜索

学习效果评测

1. 输入一个图和颜色数 k，编写一个程序，将此图的顶点涂上颜色，使得相邻两个顶点的颜色不同。

输入：

```
3          (颜色数 k)
4          (输入图中顶点的个数，并且以 1, 2, 3 …编号)
5          (输入图中连线的条数)
1 2        (以下是连线的数据)
1 4
2 3
2 4
3 4
```

输出：

```
1 1    (顶点 1 被涂成颜色 1)
2 2    (顶点 2 被涂成颜色 2)
3 1    (顶点 3 被涂成颜色 1)
4 3    (顶点 4 被涂成颜色 3)
```

2. 输入一个整数集合 S 和数字 m，编写一个程序，找出集合中的元素，使得其总和刚好为 m。

输入：

```
3 1 5 8 13 6 7    (集合 S)
18                (数字 m)
```

输出：

```
5 13
```

3. 哈密尔顿路径：输入一个图，编写一个程序，找到一条路径，刚好经过每一个顶点一次，并且回到原点。

输入：

```
5          (输入图中顶点的个数，并且以 1, 2, 3 …编号)
7          (输入图中连线的条数)
1 2        (以下是连线的数据)
1 4
2 3
2 4
3 4
3 5
4 5
```

输出：

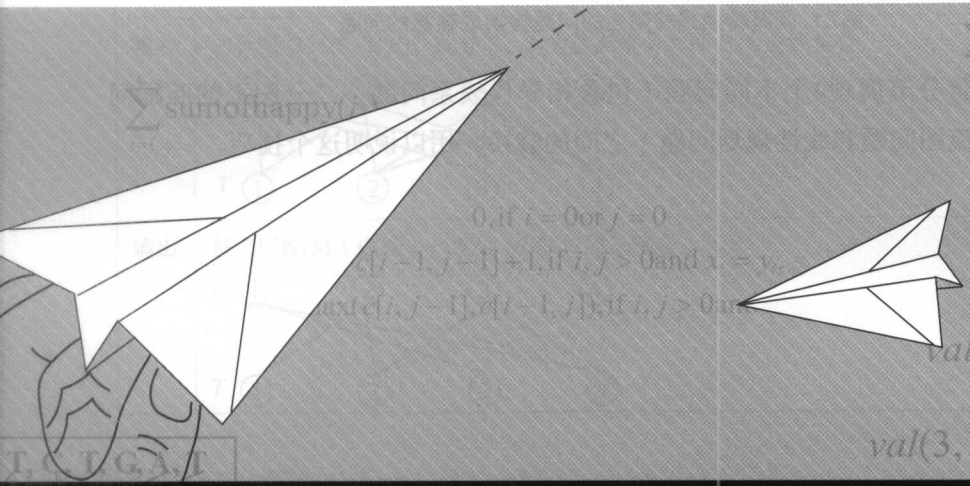
1 → 2 → 3 → 5 → 4 → 1

问题转换

章节大纲

有一位数学家和他的太太被问到一个问题：“假如你在地下室，而你想要烧开水，你会怎么做？”这位数学家说，他会上楼到厨房里烧水，他的太太也有类似的回答。现在两人再被问到一个问题：“假如你在厨房，而你想要烧水，你会怎么做？”数学家的太太说：“这简单，我会在水壶里装上水，然后开始烧水。”数学家回答说：“还有更简单的解决方法，我会先走到地下室，接下来的问题之前已经解决过了。”

笑话一则。



7.1　何谓问题转换

什么是问题转换（problem transformation）？

简而言之，就是将陌生的问题转换成熟悉的问题后，借助解决此熟悉的问题间接地解决原来陌生的问题。

问题转换是将不易求解的陌生问题转换成较熟悉的问题。因为较熟悉的问题存有已知的解法，所以可以利用此现成的解法解决此熟悉的问题，同时可以间接地找到原来陌生问题的答案。

粗略地说，问题转换是一种“借刀杀人”的伎俩，在金庸的武侠小说中有点像“乾坤大挪移”神功，可以轻易地将敌人的猛烈攻击转化于无形。图7.1表达出了这个概念，注意转换的方式（图中弯弯曲曲的线）可能有很多种。

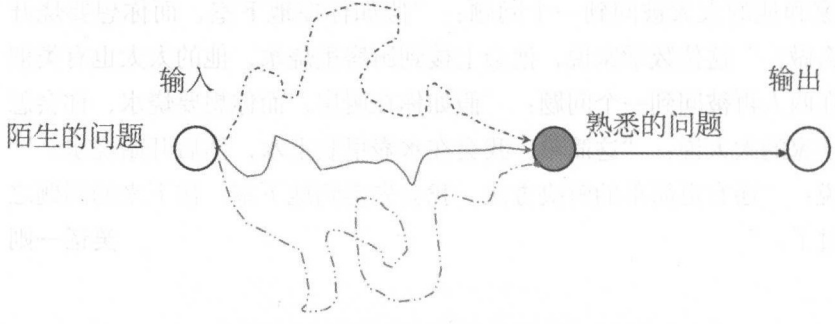


图7.1　问题转换是将陌生的问题转换成熟悉的问题

后文将分节展现4个不同问题（包含相异代表系问题、二分图上的匹配问题、网络流图问题和线性规划问题）之间的转换，用以说明这个技巧。

7.2　将相异代表系问题转换成二分图上的匹配问题

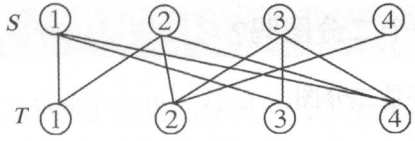
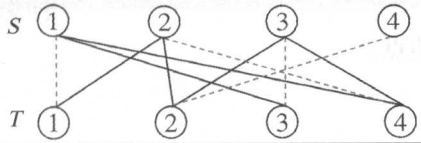
第一个例子是将相异代表系问题（systems of distinct representatives）转换成二分图上的匹配问题（matching problem in bipartite graphs）。首先解释这两个问题，再说明其中的转换。表7.1所示为相异代表系问题的说明。

表 7.1　相异代表系问题

问题	学校有若干社团，各有固定成员若干名。学校要求每个社团需选出一位会长，而且不同社团的会长不可以同一人兼任。 请你帮所有社团（任意）选出其会长，以便于社团管理正常进行。注意同一人可参加多个社团
输入	所有社团及其成员： (1)S_1={1, 2}, S_2={2, 3, 4}, S_3={1, 3}, S_4={1, 2, 3} (2)S_1={1, 2}, S_2={2, 3, 4}, S_3={1, 3}, S_4={1, 2, 3}, S_5={2, 3}
输出	(1)S_1={1, 2}, S_2={2, 3, 4}, S_3={1, 3}, S_4={1, 2, 3}，其中粗体数字代表会长 (2)无解

接下来说明二分图上的匹配问题，如表7.2所示。

表 7.2 二分图上的匹配问题

问题	一个图的顶点（vertex）可分割成两个集合 S 和 T，使得所有连线（edge）都只从 S 连接到T，称此图为二分图（bipartite graphs）。此问题是在一个二分图上选择最多的连线，使得被选出的连线没有共享的端点（endpoint）
输入	二分图 G = (S, T, E)，此处 S、T 为顶点集合，而 E 为连线集合 
输出	最大匹配 M（虚线），此处 M 为 E 的子集合 

如何将相异代表系问题转换成二分图上的匹配问题?

两个问题乍看起来一点都不像。

可以将相异代表系问题使用图来表示吗?

什么是图?

图是在几个顶点中用一条连线连起两个顶点，代表两个顶点之间的关系。

顶点有什么用途?

顶点代表你关心的事物，你关心什么?

哪些人可以当某一个社团的会长?

主要的事物是什么?

好像是“人”和“社团”。

这些事物之间的关联是什么?

哪个“人”担任这个“社团”的会长。

如果“人”和“社团”都用顶点表示，那么顶点和顶点之间的连线代表什么关系?

当某“人”担任某“社团”的会长时，连接这两个顶点。

可以画出一个图吗? 此图是一个二分图吗?

“人”和“社团”各分一边，刚好是二分图。

原来是相异代表系问题，现在是在二分图上寻找什么?

找很多连线，但是连线不可共享端点。

这是怎样的问题?

这是匹配问题，我完成了问题的转换!

以上讨论可以将相异代表系问题转换成二分图上的匹配问题。转换后，根据表7.1的输入，可以画出如图7.2所示的二分图。此图的最大匹配可以代表在社团中选出的会长名单。

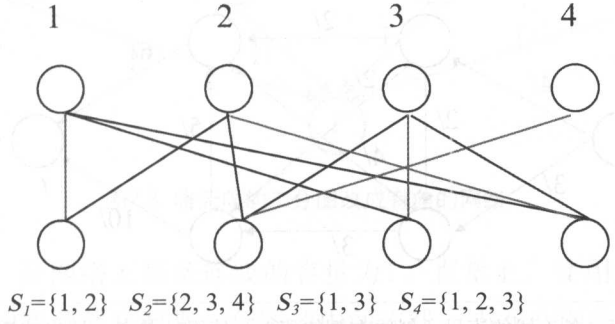


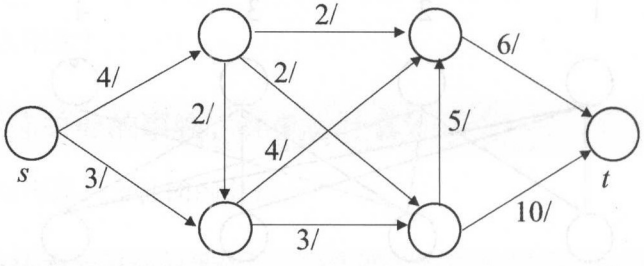
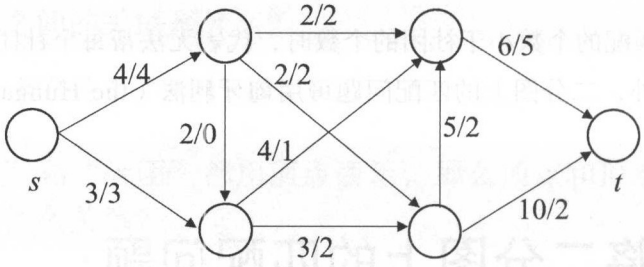
图7.2 相异代表系问题被转换成二分图上的匹配问题

当最大匹配的个数小于社团的个数时，代表无法帮每个社团选出一位专职的会长。此外，二分图上的匹配问题可用匈牙利法（the Hungarian Method）解决。

7.3 将二分图上的匹配问题转换成网络流图问题

第二个例子是将二分图上的匹配问题转换成网络流图问题（network flow problem)。首先解释网络流图问题，再说明其中的转换。表7.3所示为网络流图问题的说明。

表 7.3 网络流图问题

问题	一个运输网络是一个有向图。此网络上有一个顶点为源点（source）s、另一个顶点为汇集点（sink）t。每一条连线（edge）e上有一个值 $c(e)$（非负整数），代表容量（capacity）。每一条连线e上有另一个值$f(e)$（非负数），代表此连线的流量（flow）。一个网络流图是在每一条连线 e上决定其流量 $f(e)$，其值不可超过容量 $c(e)$；而且除了源点 s 和汇集点 t外，每一个顶点需符合流入流量的总和等于流出流量的总和 网络流图问题是找出一个网络流图，使得从源点 s 流出的流量总和最大
输入	一个有向图（含有源点 s 和汇集点 t）和每条连线的容量（斜线左侧的值） 
输出	决定网络每一条连线的流量（斜线右侧的值），使得从源点 s 流出的流量总和最大。此图最大流量总和为 3+4 = 7 

接下来，我们将二分图上的匹配问题转换成网络流图问题。首先，将匹配问题的（无方向）二分图转换成网络流图问题的有向网络。以表7.2中的匹配问题为例，我们在二分图 $G=(S, T, E)$上下分别新增两顶点 s 和 t（源点 和汇集点）。其中，源点 s 连接朝向 S 中的所有顶点，汇集点 T 中的所有顶点连接朝向顶点 t。接着，将所有原来 E 中的连线改成从 S 指向 T 的有向连线，如图7.3所示。

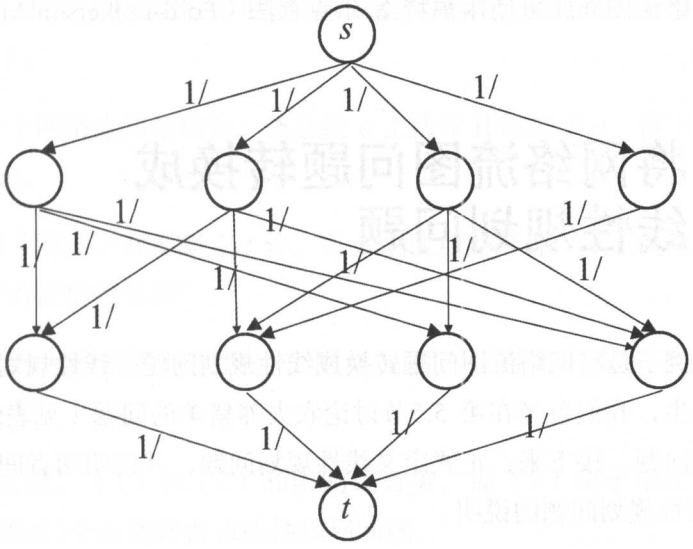


图7.3 将无向的二分图改成有向的网络

紧接着，让网络上每条连线的容量为1。直觉上，使用二分图改造的网络，其流量的瓶颈就是原来二分图的中间连线。当在此网络上找到最大流量时，必须经过二分图中的最多连线以传输最大流量，也可以间接地找到原来二分图上的最大匹配，如图7.4中的粗线。

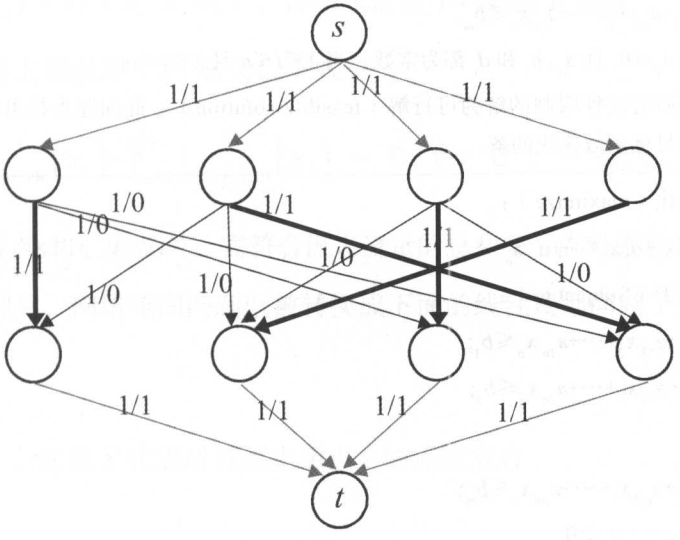


图7.4 找到最大流量(4)时，也可以找到原来二分图上的最大匹配（4 条粗线）

上述网络流图问题可使用福特-富尔克森法（Ford-Fulkerson Method）找到一个整数解。

7.4 将网络流图问题转换成线性规划问题

最后的例子是将网络流图问题转换成线性规划问题。线性规划问题对我们来说并不陌生，我们曾经在第 5.5节讨论农夫养猪羊的问题（见表5.8）和对应的线性规划问题。接下来，正式定义线性规划问题，并说明两者间的转换。表7.4所示为线性规划问题的说明。

表 7.4 线性规划问题

问题	给一个目标函数(objective function) $Z=d_1x_1+d_2x_2+\cdots+d_nx_n$ 和 m 个线性限制(linear constraint): $a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n\leqslant b_1$; $a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n\leqslant b_2$; ⋮ $a_{m1}x_1+a_{m2}x_2+\cdots+a_{mn}x_n\leqslant b_m$。 此处 $x_i\geqslant 0$, 且 a_{ji}, b_i 和 d_i 都为常数，当 $1\leqslant i\leqslant n$ 且 $1\leqslant j\leqslant m$。 满足所有线性限制的解为可行解（feasible solution）。此问题为找出所有可行解中使得目标函数优化的解
输入	极大化（maximize）： $Z=d_1x_1+d_2x_2+\cdots+d_nx_n$ 受限于（Subject to）： $a_{11}x_1+a_{12}x_2+\cdots+a_{1n}x_n\leqslant b_1$; $a_{21}x_1+a_{22}x_2+\cdots+a_{2n}x_n\leqslant b_2$; ⋮ $a_{m1}x_1+a_{m2}x_2+\cdots+a_{mn}x_n\leqslant b_m$; $x_1, x_2, \cdots, x_n\geqslant 0$
输出	满足 m 个线性限制且使得 $Z(=d_1x_1+d_2x_2+\cdots+d_nx_n)$拥有最大值的解$(x_1, x_2, \cdots, x_n)$

接下来，我们将网络流图问题转换成线性规划问题。首先，重新叙述网络流量问题如下：

（1）一个网络流图是在每一条连线 e 上决定其流量 $f(e)$，值不可超过容量 $c(e)$。

（2）除了源点 s 和汇集点 t 外，每一个顶点需符合“流入流量的总和等于流出流量的总和”。

（3）网络流图问题是找出一个网络流图，使得流出源点 s 的流量总和最大。

经观察发现，（1）和（2）和限制式有关，而（3）用于阐述目标函数。我们会逐步将这3个叙述转换成线性规划问题。

令变量 $x_1, x_2, \cdots, x_n$ 代表每一条连线的流量，常数 $c_1, c_2, \cdots, c_n$ 代表每一条连线的容量。首先，将上面的（3）转为以下目标函数式子：

$\sum_{i \in S} x_i$ 这里代表所有流出源点 s 的连线集合。

其次，将上面的叙述（1）转为以下式子：

$x_i \leqslant c_i, \forall i, 1 \leqslant i \leqslant n$，即每个流量不可超过其容量。

最后，将上面叙述（2）转为以下式子：

$\sum_{i \text{ 为流出 } v \text{ 的连线}} (x_i) - \sum_{j \text{ 为流入 } v \text{ 的连线}} (x_j) = 0, \forall v \in V - \{s, t\}$，即每一个顶点，除了源点s和汇集点t外，需符合流入流量的总和等于流出流量的总和。

稍做整理后，网络流图问题就被转换成下面的线性规划问题了。

极大化：

$\sum_{i \in S} x_i$，这里 S 代表所有流出源点 s 的连线集合。

受限于：

（1）$x_i \leqslant c_i, \forall i, 1 \leqslant i \leqslant n$。

（2）$\sum_{i \text{ 为流出 } v \text{ 的连线}} (x_i) - \sum_{j \text{ 为流入 } v \text{ 的连线}} (x_j) = 0, \forall v \in V - \{s, t\}$。

（3）$x_i \geqslant 0, \forall i, 1 \leqslant i \leqslant n$。

如同前面两次的转换，线性规划问题也可以被单纯形法（Simplex Method）所解。

7.5　问题转换的技巧

问题转换的技巧是将一个陌生问题转换成较熟悉的问题。使用较熟悉问题早已存在的解法间接地解决此陌生问题，如图7.5所示。

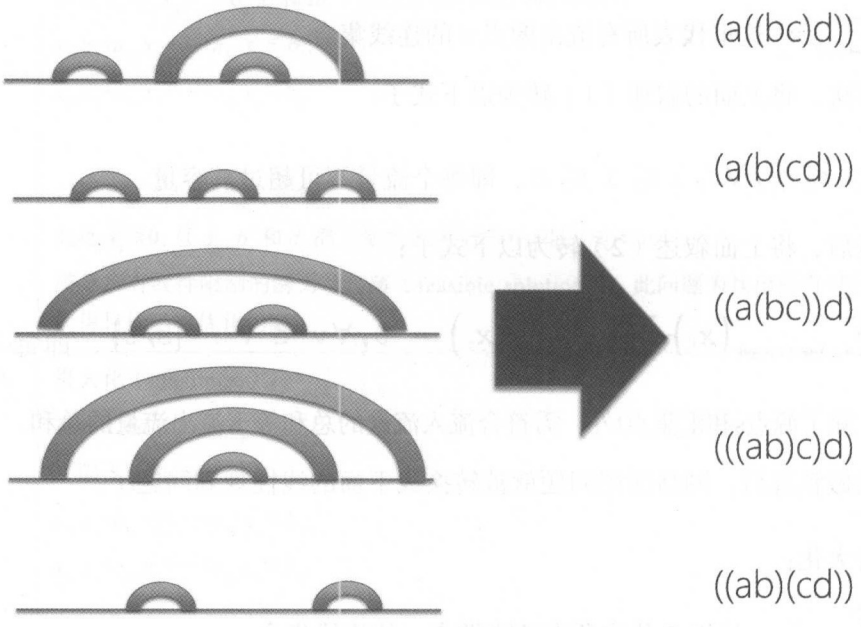


图7.5　3个贴地铁环（使得彼此不相交）的摆放问题可以转换成在4个不同符号上用括号正确分隔的问题

当我们面对一个陌生问题时，常常会问：“该转换到哪一个问题呢？”

如果你熟悉的问题本来就不多，上面的问话就难有答案了；反之，如果你常常比较两个问题间的相似处和不同处，应该会对这两个问题产生更深的体会，如图7.6所示。

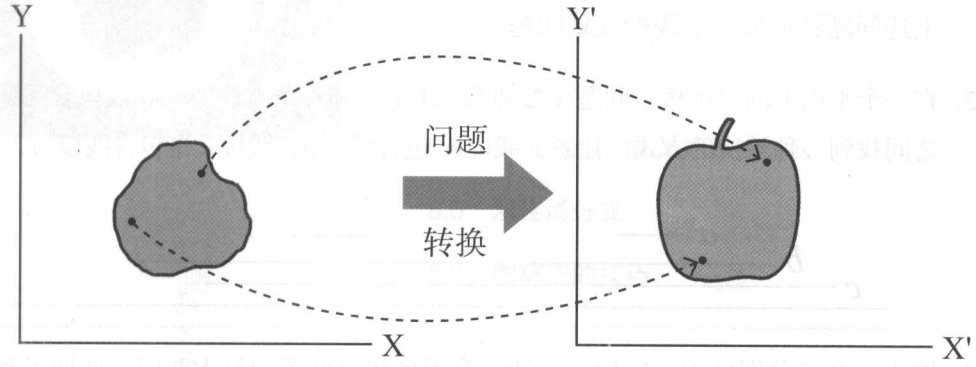


图7.6 比较相似处和不同处

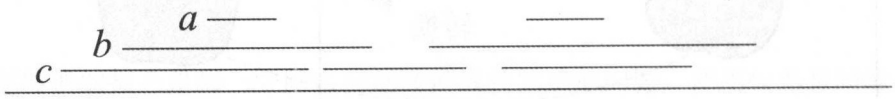
最后，引用波利亚先生（G. Polya）在《如何解题》（How to Solve It）中的一段话，作为本章的结束语：

> “你以前看过这个问题吗？你看过相似的问题吗？你看过相关的问题吗？”
>
> “这里有一个相关而且知道解法的问题。你能使用它吗？你能使用它的结果吗？你能使用它的方法吗？你可以引入一些辅助元素以便使用它吗？”
>
> “你可以重新表达这个问题吗？你可以用不同的方式表达这个问题吗？”

学习效果评测

1. 有 n 个慈善捐款人想要捐钱给 k 个孤儿院。每个捐款人想要捐款的额度有一定的上限，而且针对每个孤儿院，也有一个最高的捐款额度。为了公平起见，每个孤儿院接受捐款的总额也被设定了一个最高额度，以免独占所有捐款。此问题在于找出一个捐款的方式，使得整体的捐款达到最高。将上述问题转换成一个线性规划问题。

2. 在一个平面上的二维极点问题（参阅第 2.4 节）和一条直线上的线段包含问题之间找到一种转换的方式。注意，线段 b 包含线段 a，线段 c 也包含线段 a。



3. 输入一个二分图$G=(S, T, E)$，设计一个程序找出此图的最大匹配。此题可利用匈牙利法解决。

输入：

```
4                (顶点集合S的个数，并且以 1, 2, 3 …编号)
4                (顶点集合T的个数，并且以 1, 2, 3 …编号)
10               (连线集合 E 的条数)
1 1              (以下为连线的数据)
1 3
1 4
2 1
2 2
2 4
3 2
3 3
3 4
4 2
```

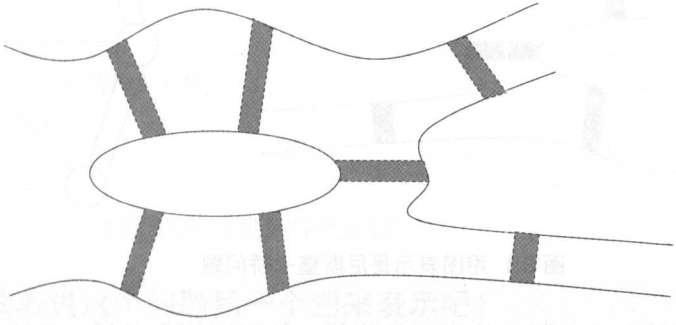
输出：

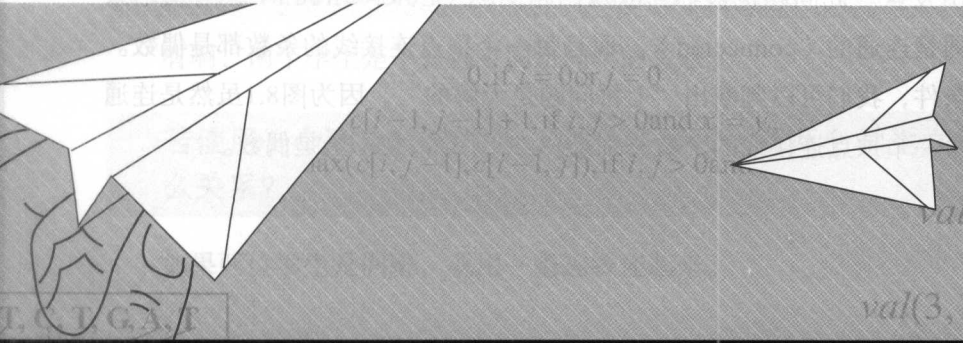
```
4                (最大匹配的个数)
1 1              (以下为匹配的数据)
2 4
3 3
4 2
```

图算法

章节大纲

在古代东普鲁士（Eastern Prussia）哥尼斯堡（Koenigsberg），有一条河流环绕keiphof 岛，并将附近土地分割成4个区域。这些区域的居民借助7座桥进行往来。当地的居民想从某一个区域刚好走过7座桥一次，并回到原出发地，你能帮他们找出这条路径吗?





8.1　什么是图

什么是图（graphs）？

简而言之，就是一个图包含几个顶点和几条连线（或称为边），每一条连线连接两个顶点。

图算法就是将问题的输入表达成一个图，并在此图上设计算法，以找到问题的输出（解）。

例如，在思考哥尼斯堡（Koenigsberg）问题时（这个问题也称为哥尼斯堡七桥问题，在图论中简称七桥问题），把隔离的4个区域当作顶点（vertex），并将桥当作连线（edge，或称为边），则柯尼斯堡的地图可以用图8.1所示的图表示。

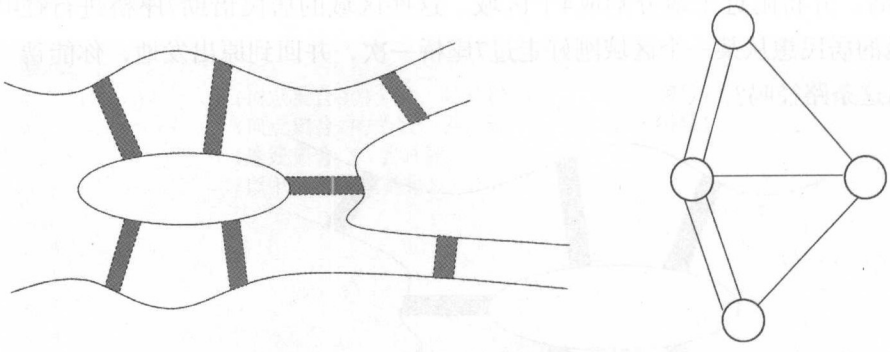


图 8.1　用图表示哥尼斯堡七桥问题

七桥问题就是在这个图上找到一条路径，使得刚好走过每一条连线一次，且回到原出发点。此问题也被称为欧拉回路问题（Euler Circuit），有解的条件是：此图是连通的（connected），而且每一个顶点连接线的条数都是偶数。依据这个条件，我们可以判断出“七桥问题是无解的”，因为图8.1虽然是连通的，但是每一个顶点连接线的个数分别是{3, 5, 3, 3}，并不全是偶数。

8.2 连通分支

第一个图算法的例子是利用找出一个图连通分支（connected components）的技巧解决问题。表8.1将说明第一个例子。

表 8.1 同班同学

问题	有一个学校拥有 N 位学生，有些学生彼此是同班同学。依据一个单纯形法，则“我的同班同学的同班同学也是我的同班同学”；也就是，如果A和B是同班同学，而且B和C是同班同学，那么A和C也是同班同学；换言之，A、B、C都是同一班的学生。当输入所有学生之间的这种关系后，计算全校共有几个班级
输入	输入的第一行是整数N，代表学校所有学生集合为{1, 2, …, N }。第二行是整数M，代表有M对同班的关系。接下来，M行同班同学的关系列在其后 6 5 1 2 3 4 4 5 4 6 5 6
输出	全校的班级个数 2

如何利用图算法解决同班同学的问题？

先试着将这个问题用一个图来表示吧！

此问题是否提到两个不同个体之间的关系？

有啊，两个学生是不是拥有同班同学的关系。

若使用一个顶点代表学生，则一条连线代表两个学生之间的什么关系？

如果两位学生是同班，就用一条连线连起来。

使用这个方法就可以将上述输入表示成图8.2所示的样子。

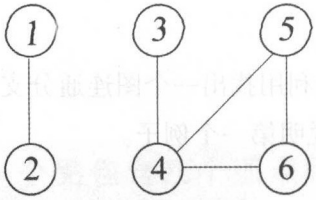


图8.2 用图表示同班同学的问题（一个顶点代表一个学生，如果两位学生是同班，就用一条线连起来）

接下来，我们要在这个图上找什么？哪些学生是同一班的？总共有多少班？从这个图上看得出总共有多少班吗？

两个班，因为有两组学生。

怎样找到每一组的学生？

从一个顶点开始，寻找所有连接得到的顶点。

对一个图进行搜索的常见方法是什么？

好像学过，是深度搜索法吗？

直觉上，图8.2中的每一组就是此图上的连通分支（connected component），即此图上最大可连通子图。如果能找到一个图连通分支的个数，就可以解决同班同学的问题。

根据以上讨论，我们可以用深度优先搜索法（Depth-First Search，DFS）展开每一个连通分支的搜索。深度优先搜索法的特点是：最晚被拜访的顶点的邻居比其他较早被拜访的顶点的邻居优先被搜索。以图8.2为例，可能的搜索顺序为 1（起始点）→2（1的邻居）→发现第一个连通分支; 3（起始点）→4（3的邻居）→5（4的邻居）→6（5的邻居）→发现第二个连通分支，如图8.3所示。详细的算法如表8.2所示。

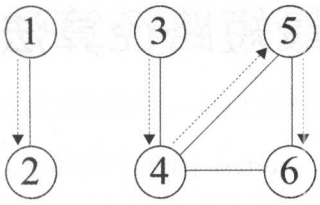


图8.3　使用深度优先搜索法开始每一个连通分支的搜索（虚线表示搜索顺序）

表 8.2　深度优先搜索法

输入	一个图 $G=(V, E)$和搜索起始点 x
输出	从起始点 x 开始，以深度优先搜索法拜访所有 V 中连接得到的点

步骤	Algorithm depth_first_search (x)
	/*图 G 有 n 个点，即 |V|=n*/
	/*矩阵 visited[1:n]全部为 false*/
	{
	Step 1: 输出 x 并令 visited [x]:=true　　/*拜访点 x*/
	Step 2: for 每一个 x 的邻居 y do
	{
	if visited [y]:=false
	then depth_first_search (y)　　/*递归地拜访顶点 y*/
	}
	}

最后，使用深度优先搜索法逐一搜索每一个连通分支。表8.3是使用图算法解决同班同学问题的连通分支算法。

表 8.3 连通分支算法

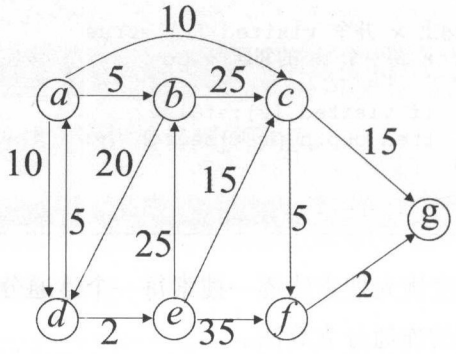
输入	图 $G=(V, E)$
输出	图 G 的连通分支数目

步骤	Algorithm connected_component (G)
	/*图 G 有 n 个，即 |V|=n*/
	/*令矩阵 visited[1:n]全部为 false*/
	{
	Step 1: y:=0;　　/*连通分支的数目一开始为 0*/
	Step 2: for i:=1 to n do
	if visited [i]:=false then
	{depth_first_search(i);/*归地找出点i可连接到的一个连通分支*/
	y:=y+1;　　/*增加一个连通分支*/
	}
	Step 3: 输出连通分支数目 y
	}

8.3　Dijkstra 最短路径算法

第二个例子是最短路径（shortest paths）的问题，如表8.4所示。

表 8.4 最短路径问题

问题	小王考虑从甲地开车前往其他城市，但不知道需要花费多少交通时间才可到某一个城市。现在有一张地图记载了所有城市的位置，以及两两城市之间所需要的交通时间。 请使用此地图找到一条从甲市出发，最终可以抵达另一个城市（每一个）的最短路径（所需要的交通时间总和最短）
输入	一个图 $G=(V, E)$ 和起始点（出发地）a。在此图中，用顶点代表城市，用一条连线 $e=(v_1, v_2)$和连线上的数字 $t[v_1, v_2]$ 代表有一条路从城市 v_1 连接另一个城市 v_2，开车需要 $t[v_1, v_2]$ 的时间（此值非负数，即 $t[e]\geqslant 0$） 
输出	a 到其他每一个顶点的最短路径： $a \rightarrow a$，时间 0 $a \rightarrow b$，时间 5 $a \rightarrow c$，时间 10 $a \rightarrow d$，时间 10 $a \rightarrow d \rightarrow e$，时间 12 $a \rightarrow c \rightarrow f$，时间 15 $a \rightarrow c \rightarrow f \rightarrow g$，时间 17

如果能将地图上的交通信息转成一个图，再找出此图上从源点到每一个顶点的最短路径，就可以解决此问题。在此我们称两顶点的距离（distance）为这两顶点间最短路径上所有连线时间的总和。

8.3.1 直观概念——Dijkstra 最短路径算法

将表8.4中的图从起始点到其他每一个顶点的最短路径按照距离长度从小到大列出如下：

```
a→a，时间 0
a→b，时间 5
a→c，时间 10
a→d，时间 10
a→d→e，时间 12
a→c→f，时间 15
a→c→f→g，时间 17
```

最短路径之间有什么关系？

最短路径的前面部分也是一条最短路径。例如，a→c→f→g 的前面部分为 a→c→f，a→c→f 的前面部分为 a→c，都是最短路径。

哪一个距离较长？

包含较多顶点的路径距离较长。

为什么？

因为每两个顶点间所需的时间是正的，多走一段路就多需一些时间。

如果从短到长找出所有最短路径，那么这些最短路径有什么关系？

除了最后一个顶点外，其他顶点的最短路径会比较早被找到。

如何选择最短路径的最后一点？

不知道，大概是找最短路径吧。

根据以上讨论，Dijkstra 最短路径算法是由近而远地找出起始点 a 到其他每一个顶点的最短路径。此算法一直记录当前找到最短路径的顶点集合 S，并使用经过 S 的顶点向外跨一步，找到下一个距离起始点最近的最短路径。

更精确地说，Dijkstra 最短路径算法尝试找到从顶点 a出发，只经过 S 中的顶点，最后抵达 S 的外围邻居 u 的一条最短的路。注意，u 并不在 S 集合中，但是距离 S 中的一个顶点只有一步。因此，为找到下一条最短路径（及靠近顶点 a 的顶点），需计算顶点 a（只经过 S 中的顶点）到每一个 S 的外围邻居 u 的路径值，并将此值存储于 dist [u]中。此算法会从所有可能的这种顶点 u 中选择一个拥有最小 dist [u]值的顶点，而 dist[u] 是其最短路径。最后将顶点 u 纳入集合 S 中。

Dijkstra 最短路径算法重复以上操作，直到所有顶点纳入集合 S 中为止。

总结以上讨论，这些特性有助于了解 Dijkstra 最短路径算法：令S 集合包括当前已经找到距离最靠近 a 的顶点（含起始点 a）。若下一个最靠近 a 的顶点为 u，从 a 走到 u 的最短路径所有中途经过的顶点都会在 S 中。

直觉上，因为每一步都需要花费额外的移动时间，所以越靠近起始点的顶点越被先发现。如图8.4所示，虚线内的顶点（代表 S）是当前找到最靠近 a 的顶点。假设下一个最靠近 a 的顶点是从 a 走到 u 的路径，那么除了此路径的最末顶点 u 外，不会经过虚线外的顶点。否则，假设此路径中尚有另一个顶点 w 落在 S 外，那么从 a 走到 w 的路径将会比从 a 走到 u 的路径短（因为从 w 走到 u 的路径所需时间必为正数）。但是，如此又违反了“顶点 u 是下一个最靠近 a 的顶点”的假设。结论是此特性是正确的。

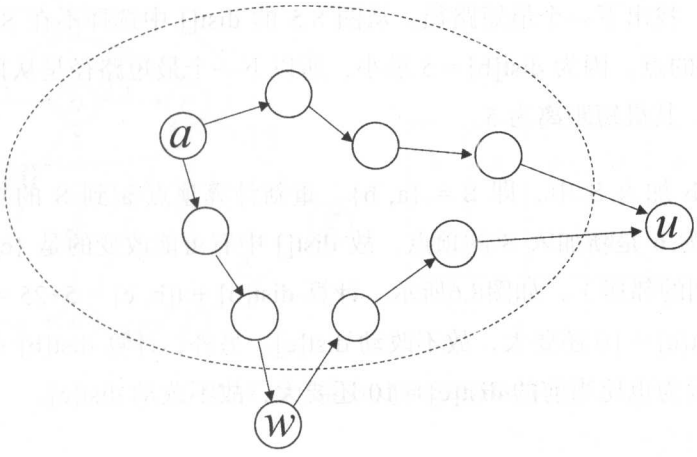
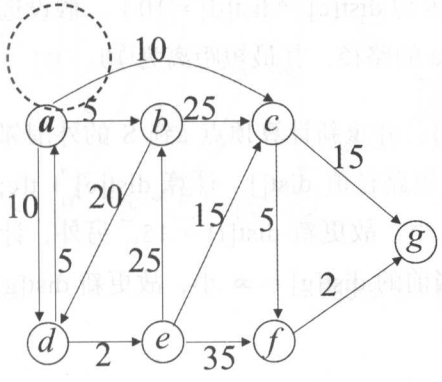


图 8.4　下一个最靠近 a 的顶点，其路径除了最后一个顶点外，不会经过虚线外的顶点

8.3.2　范例——Dijkstra 最短路径算法

下面将以表8.4中的图为例子进行说明。

起初，S 只包含顶点 a，即 S = {a}。计算顶点 a 直接走一步的最短距离。因为顶点 a 可连接到 b（距离 5），可连接到 c（距离 10），也可连接到 d（距离 10），所以相对应的 dist [] 值可计算出，其余的 dist [] 值暂时为 ∞，如图8.5所示。

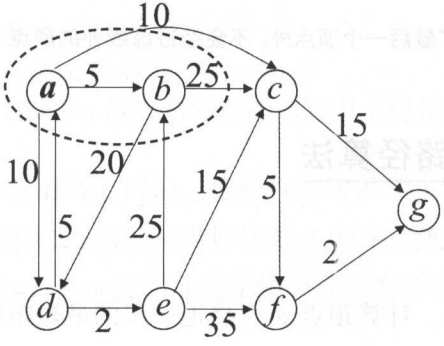


a	b	c	d	e	f	g
0	5	10	10	∞	∞	∞

图8.5　S = {a} 和 dist[]

接下来，找出下一个最短路径。从图 8.5 的 dist[] 中选择不在 S = {a}中且 dist[] 值最小的点。因为 dist[b] = 5 最小，所以下一个最短路径是从顶点 a 到顶点 b 的路径，其最短距离为 5。

此刻将 b 加入 S 中，即 S = {a, b}。重新计算顶点 a 到 S 的外围邻居的 dist[]值。因为 b 是新加入 S 的顶点，故 dist[] 中有可能改变的是 {c, d}（顶点 b 直接连接到的邻居），如图8.6所示。计算 dist[b] + t[b, c] = 5+25 = 30，因为比当前的 dist[c] = 10 还要大，故不改动 dist[c]。另外，计算 dist[b] + t[b, d] = 5 + 20 = 25，因为也比当前的 dist[c] = 10 还要大，故不改动 dist[c]。

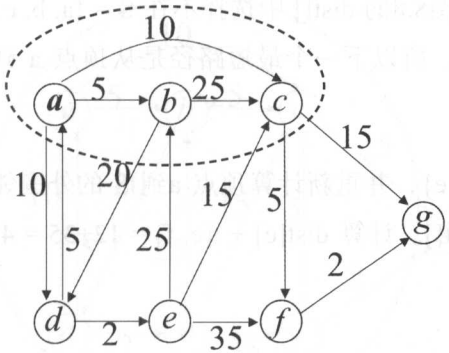


a	b	c	d	e	f	g
0	5	10	10	∞	∞	∞

图8.6 S = {a, b}和 dist[]

接下来，找出下一个最短路径。从图8.6的 dist[] 中 选择不在 S = {a, b} 中且 dist[] 值最小的顶点（即顶点 c 或 d，因为 dist[c] = dist[d] = 10）。假设选择 c，下一个最短路径就是从顶点 a 到顶点 c 的路径，其最短距离为 10。

此刻将 c 加入 S 中，即 S = {a, b, c}，并重新计算顶点 a到 S 的外围邻居 {f, g}（顶点 c 直接连接到的邻居）的最短路径值 dist[]。计算 dist[c] + t[c, f] = 10+5 = 15，因为比当前的 dist[f] = ∞ 小，故更新 dist[f] = 15。另外，计算 dist[c] + t[c, g] = 10+15 = 25，因为也比当前的 dist[g] = ∞ 小，故更新 dist[g] = 25，如图8.7所示。

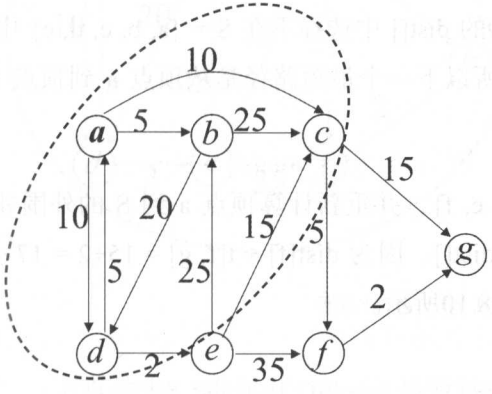


a	*b*	*c*	*d*	*e*	*f*	*g*
0	5	10	10	∞	15	25

图8.7 S = {a, b, c}和 dist[]*dist*[]

接下来继续完成后续的步骤。从图8.7的 dist[] 中选择不在 S = {a, b, c} 中且 dist[] 值最小的顶点（即顶点 d）。所以下一个最短路径是从顶点 a 到顶点 d 的路径，其最短距离为 10。

此刻将 d 加入 S 中，即 S = {a, b, c, d}，并重新计算顶点 a 到 S 的外围邻居{e}（顶点 d 直接连接到的邻居）的最短路径值 dist[]。计算 dist[d] + t[d, e] = 10+2 = 12，因为比当前的 dist[e] = ∞ 小，故更新 dist[e] = 12，如图8.8所示。

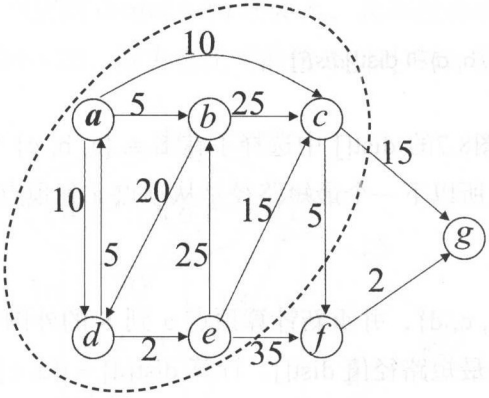


a	*b*	*c*	*d*	*e*	*f*	*g*
0	5	10	10	12	15	25

图8.8 S = {a, b, c, d} 和 dist[]

接下来，寻找下一个最短路径。从图8.8的 dist[] 中选择不在 S = {a, b, c, d} 中且 dist[] 值最小的顶点（即顶点 e）。所以下一个最短路径是从顶点 a 到顶点 e 的路径，其最短距离为 12。

将 e 加入 S 中，即 S = {a, b, c, d, e}，并重新计算顶点 a到 S 的外围邻居 {f}（顶点 e 直接连接到的邻居）的 dist[]。计算 dist[e] + t[e, f] = 12+35 = 47 > dist[f] = 15，故不更新，如图8.9所示。

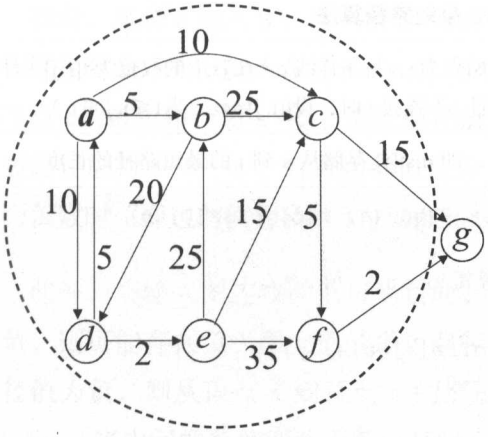


a	b	c	d	e	f	g
0	5	10	10	12	15	25

图8.9 S = {a, b, c, d, e} 和 dist[]

继续寻找下一个最短路径。从图8.9的 dist[] 中选择不在 S = {a, b, c, d, e} 中且 dist[] 值最小的顶点（即顶点 f）。所以下一个最短路径是从顶点 a 到顶点 f 的路径，其最短距离为 15。

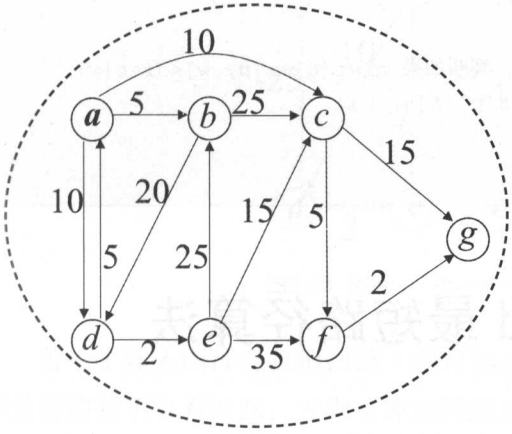
将 f 加入 S 中，即 S = {a, b, c, d, e, f}，并重新计算顶点 a 到 S 的外围邻居{g}（顶点 f 直接连接到的邻居）的 dist[]。因为 dist[f] + t[f, g] = 15+2 = 17 < dist[g] = 25，故更新 dist[g] = 17，如图8.10所示。



a	b	c	d	e	f	g
0	5	10	10	12	15	17

图8.10 S = {a, b, c, d, e, f } 和 dist[]

最后，将 g 加入 S 中，即 S = {a, b, c, d, e, f, g}，并结束此算法，如图 8.11 所示。



a	b	c	d	e	f	g
0	5	10	10	12	15	17

图8.11 S = {a, b, c, d, e, f, g} 和 dist[]

下页的表8.5将列出 Dijkstra 最短路径算法。

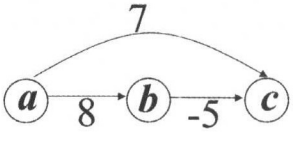
表 8.5 Dijkstra 最短路径算法

输入	一个图 G = (V={1, 2, 3, ⋯, n }, E)，起始点为a，每条连线 e = (i, j) 上的权重为 t[i, j]（注意，权重不可为负值，即 t[i, j] ≥0；且 i 不连接 j 时，设t[i, j]=∞，当1≤i, j ≤n）
输出	a 到其他每一个顶点的最短路径长度，即 dist[j] 存储从 a 到 j 的最短路径的长度

步骤	Algorithm Dijkstra_Shortest_Path (*a*, *t*[], *dist*[], *n*)
	{
	/* 将起始点 a 选入 S 中，并设置其 dist[] 值 */
	Step 1: *S*={*a*};
	当 1≤*j*≠*a*≤*n* 时，令 *dist*[*j*]=*t*[*a*, *j*];
	当 *j*=*a* 时，令 *dist*[*j*]=0;
	
	/* 决定从起始点 *a* 出发到达其他顶点的*n*-1 条最短路径 */
	Step 2: 自 *i*=2 到 *n*-1 执行下列指令
	{
	/* 找到下一个最短路径 */
	从 *V-S* 中找出 *u*，使其 *dist*[*u*]值为最小;
	将 *u* 加入 *S* 中;
	
	/* 更新 *dist*[] */
	针对*V-S*中的每一顶点w，判断如果 *dist*[*u*]+*t*[*u*, *w*]<*dist*[*w*]，
	就将*dist*[*w*]设为 *dist*[*u*]+*t*[*u*, *w*];
	}
	}

8.4 Bellman-Ford 最短路径算法

当输入图连线的值为负数时（即边的值为负数），Dijkstra 最短路径算法有可能找不到最短路径。如图8.12所示，令 a 为起始点，起初 S 只包含顶点 a（即 S = {a}）。计算顶点 a到 S 的外围邻居 {b, c}的 dist[] 值。



a	*b*	*c*
0	8	7

图8.12 S = {a} 和 dist[]

接着，选择 c 加入 S，即当前 S = {a, c}，dist[] 的值并未改变。根据 Dijkstra 最短路径算法，当 c 加入 S 后，dist[c] 将不再被改变。此时 dist[c] = 7 代表顶点 a 到顶点 c的最短路径为 a→c，而此路径值为 7，但是在此图中顶点 a到顶点 c 的最短路径应为 a→b→c，且其路径值应为 8-5 = 3 才对。因此，当输入图连线的值为负数时（即边的值为负数），Dijkstra 最短路径算法有可能出错。

此外，当输入图连线的值可为负时，若存在一个回路（cycle），其路径值为负，则可能存在负无限大值的最短路径。如图8.13所示，路径 a→b→d→a 的路径值为负，则从顶点 a 到顶点 c 的最短路径可为 a→b→d→a→b→d→a→…→b→c。当中间的负值回路不断重复时，顶点 a 到顶点 c 的最短路径值可为负无限大。

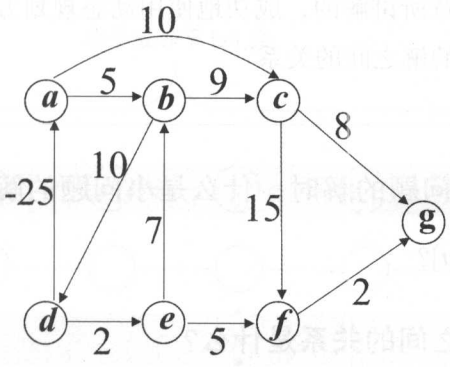


图8.13 回路 a→b→d→a 的路径值为负数

若一个输入图没有负值回路（但可能有负值的连线），则图中任意两顶点的最短路径不会有回路；否则去掉此回路后，反而会得到另一条更短的路径。

8.4.1 直观概念——Bellman-Ford 最短路径算法

下面介绍 Bellman-Ford 最短路径算法，可以弥补 Dijkstra 最短路径算法不能处理负值连线的缺憾。简单地说，Bellman-Ford 最短路径算法的精神是“动态规划”。

首先，需要两个符号：

$dist[u]$ 代表自起始点 a 到顶点 u 的最短路径值。

$dist_k[u]$ 代表自起始点 a 到顶点 u 的最短路径值，但是只经过最多k条连线（或边）。

当输入图为 G = (V={1, 2, 3, …, n}, E)时，每条连线 e = (i, j) 上的权重为 t[i, j]（注意，此处权重可为负值，且当 i 不连接 j 时，设 t[i, j] = ∞，1≤i, j≤n）。此时，dist1[u] = t[a, u]，1≤u≤n（因为初始时，只有一条连线可经过）。并且一条最短路径最多只有 n-1 条连线（否则会造成回路），distn-1[u] 就等于从起始点 a 到顶点 u 的最短路径值 dist[u]。

Bellman-Ford 最短路径算法就是利用动态规划的方式计算所有顶点的 distn-1[u] 值。如第3章所讲解的，成功地使用动态规划方法的关键在于“找出大问题的解和小问题的解之间的关系”。

当 $dist_k[u]$是大问题的解时，什么是小问题的解？

会不会是 $dist_{k-1}[u]$？

如果是，他们之间的关系是什么？

嗯！不知道。

$dist_k[u]$和 $dist_{k-1}[u]$分别代表什么？

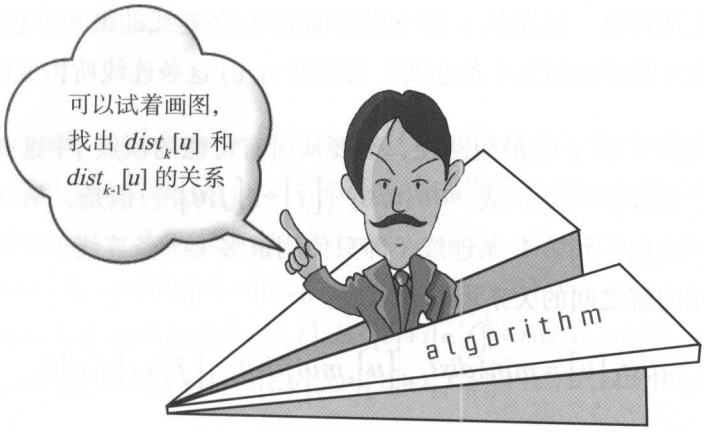
$dist_k[u]$是从起始点 a 到顶点 u只经过最多 k条连线的最短路径值，而 $dist_{k-1}[u]$是从起始点a 到顶点 u只经过最多 k-1条连线的最短路径值。

它们之间的关系是什么？有什么不同？

$dist_k[u]$的路径比 $dist_{k-1}[u]$的路径多一条连线。

可以利用 $dist_{k-1}[u]$找到 $dist_k[u]$吗？

“好难想象呀！”



$dist_k[u]$和$dist_{k-1}[u]$的关系如图8.14所示。

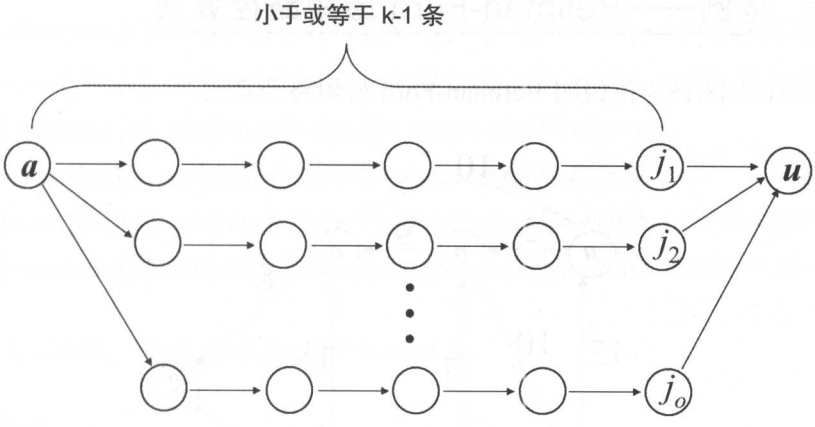


图8.14　寻找 $dist_k[u]$ 和 $dist_{k-1}[u]$ 的关系

从*a* 到 *u* 只经过最多 *k* 条连线的最短路径中，有没有藏着一个类似问题的解？

在此最短路径中，从 *a* 到 *u* 的前面一个顶点 j 的路径应该是最多使用 k-1 条连线的最短路径。

这个顶点 *j* 是哪一个顶点？

好像很多顶点都有可能。只要从 *a* 出发经过最多 *k*-1 条连线，最后连接顶点 *j*，顶点 *j* 又可连接顶点 *u* 即可。

基于上述讨论，这条从 a 到 u 最多使用 k 条连线的最短路径，是从a 到 j 的最短路径（最多经过 k-1 条连线）再衔接 (j, u) 这条连线所构成的。

为了找到 a 到 u 的最短路径，需要从所有可能的顶点 j中选择其中最短的一条路径。此步骤可表示成 $\min_j\{dist_{k-1}[j]+t[j,u]\}$。但是，从 a 到u 的最短路径也有可能用不到第 k 条连线（即只使用最多 k-1 条连线），所以大问题的解和小问题的解之间的关系可以表示成：

$$dist_k[u]=min\{dist_{k-1}[u],\min_j\{dist_{k-1}[j]+t[j,u]\}\}$$

利用此关系，Bellman-Ford 最短路径算法就可被设计出来了。

8.4.2　范例——Bellman-Ford 最短路径算法

下面以图8.15为例说明 Bellman-Ford 最短路径算法。

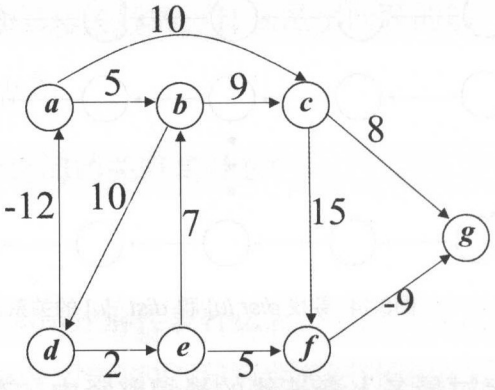


图8.15 Bellman-Ford 最短路径算法的范例

当 k =1 时，设置从起始点 a 仅利用一条连线可抵达的顶点（即 b, c）的路径值，其他顶点因使用一条连线到不了，故设置其值为 ∞。

$dist_1[a]=t[a,a]=0$, $dist_1[b]=t[a,b]=5$, $dist_1[c]=t[a,c]=10$, $dist_1[d]=t[a,d]=\infty$, $dist_1[e]=t[a,e]=\infty$, $dist_1[f]=t[a,f]=\infty$, $dist_1[g]=t[a,g]=\infty$。

当 k =2 时，利用 dist1 [] 计算出 dist2 []。

$$dist_2[a]=0$$
$$dist_2[b]=min\{dist_1[b],\ min\{dist_1[a]+t[a,b],dist_1[e]+t[e,b]\}\}=min\{5,\ min\{0+5,\infty+7\}\}=5$$
$$dist_2[c]=min\{dist_1[c],\ min\{dist_1[a]+t[a,c],dist_1[b]+t[b,c]\}\}=min\{10,\ min\{0+10,5+9\}\}=10$$
$$dist_2[d]=min\{dist_1[d],\ min\{dist_1[b]+t[b,d]\}=min\{\infty,\ min\{5+10\}\}=15$$
$$dist_2[e]=min\{dist_1[e],\ min\{dist_1[d]+t[d,e]\}=min\{\infty,\ min\{\infty+2\}\}=\infty$$
$$dist_2[f]=min\{dist_1[f],\ min\{dist_1[c]+t[c,f],dist_1[e]+t[e,f]\}=min\{\infty,\ min\{10+15,\infty+5\}\}=25$$
$$dist_2[g]=min\{dist_1[g],\ min\{dist_1[c]+t[c,g],dist_1[f]+t[f,g]\}=min\{\infty,\ min\{10+8,\infty-9\}\}=18$$

当 k =3 时，利用 dist2 [] 计算出 dist3 []。

$$dist_3[a]=0$$
$$dist_3[b]=min\{dist_2[b],\ min\{dist_2[a]+t[a,b],dist_2[e]+t[e,b]\}\}=min\{5,\ min\{0+5,\infty+7\}\}=5$$
$$dist_3[c]=min\{dist_2[c],\ min\{dist_2[a]+t[a,c],dist_2[b]+t[b,c]\}\}=min\{10,\ min\{0+10,5+9\}\}=10$$
$$dist_3[d]=min\{dist_2[d],\ min\{dist_2[b]+t[b,d]\}=min\{15,\ min\{5+10\}\}=15$$
$$dist_3[e]=min\{dist_2[e],\ min\{dist_2[d]+t[d,e]\}=min\{\infty,\ min\{15+2\}\}=17$$
$$dist_3[f]=min\{dist_2[f],\ min\{dist_2[c]+t[c,f],dist_2[e]+t[e,f]\}=min\{25,\ min\{10+15,\infty+5\}\}=25$$
$$dist_3[g]=min\{dist_2[g],\ min\{dist_2[c]+t[c,g],dist_2[f]+t[f,g]\}=min\{18,\ min\{10+8,25-9\}\}=16$$

当 k = 4 时，利用 dist3 [] 计算出 dist4 []。

$$dist_4[a]=0$$
$$dist_4[b]=min\{dist_3[b],\ min\{dist_3[a]+t[a,b],dist_3[e]+t[e,b]\}\}=min\{5,\ min\{0+5,17+7\}\}=5$$
$$dist_4[c]=min\{dist_3[c],\ min\{dist_3[a]+t[a,c],dist_3[b]+t[b,c]\}\}=min\{10,\ min\{0+10,5+9\}\}=10$$
$$dist_4[d]=min\{dist_3[d],\ min\{dist_3[b]+t[b,d]\}=min\{15,\ min\{5+10\}\}=15$$
$$dist_4[e]=min\{dist_3[e],\ min\{dist_3[d]+t[d,e]\}=min\{17,\ min\{15+2\}\}=17$$
$$dist_4[f]=min\{dist_3[f],\ min\{dist_3[c]+t[c,f],dist_3[e]+t[e,f]\}=min\{25,\ min\{10+15,17+5\}\}=22$$
$$dist_4[g]=min\{dist_3[g],\ min\{dist_3[c]+t[c,g],dist_3[f]+t[f,g]\}=min\{16,\ min\{10+8,25-9\}\}=16$$

当 k = 5 时，利用 dist4 [] 计算出 dist5 []。

$dist_5[a]=0$

$dist_5[b]=min\{dist_4[b],\ min\{dist_4[a]+t[a,b],dist_4[e]+t[e,b]\}\}=min\{5,\ min\{0+5,17+7\}\}=5$

$dist_5[c]=min\{dist_4[c],\ min\{dist_4[a]+t[a,c],dist_4[b]+t[b,c]\}\}=min\{10,\ min\{0+10,5+9\}\}=10$

$dist_5[d]=min\{dist_4[d],\ min\{dist_4[b]+t[b,d]\}=min\{15,\ min\{5+10\}\}=15$

$dist_5[e]=min\{dist_4[e],\ min\{dist_4[d]+t[d,e]\}=min\{17,\ min\{15+2\}\}=17$

$dist_5[f]=min\{dist_4[f],\ min\{dist_4[c]+t[c,f],dist_4[e]+t[e,f]\}=min\{22,\ min\{10+15,17+5\}\}=22$

$dist_5[g]=min\{dist_4[g],\ min\{dist_4[c]+t[c,g],dist_4[f]+t[f,g]\}=min\{16,\ min\{10+8,22-9\}\}=13$

当 k = 6 时，利用 dist5 [] 计算出 dist6 []。

$dist_6[a]=0$

$dist_6[b]=min\{dist_5[b],\ min\{dist_5[a]+t[a,b],dist_5[e]+t[e,b]\}\}=min\{5,\ min\{0+5,17+7\}\}=5$

$dist_6[c]=min\{dist_5[c],\ min\{dist_5[a]+t[a,c],dist_5[b]+t[b,c]\}\}=min\{10,\ min\{0+10,5+9\}\}=10$

$dist_6[d]=min\{dist_5[d],\ min\{dist_5[b]+t[b,d]\}=min\{15,\ min\{5+10\}\}=15$

$dist_6[e]=min\{dist_5[e],\ min\{dist_5[d]+t[d,e]\}=min\{17,\ min\{15+2\}\}=17$

$dist_6[f]=min\{dist_5[f],\ min\{dist_5[c]+t[c,f],dist_5[e]+t[e,f]\}=min\{22,\ min\{10+15,17+5\}\}=22$

$dist_6[g]=min\{dist_5[g],\ min\{dist_5[c]+t[c,g],dist_5[f]+t[f,g]\}=min\{13,\ min\{10+8,22-9\}\}=13$

最后，$dist_6$[]中的每一个值代表所有的最短路径值，如表8.6所示。例如，$dist_6[g]$代表顶点 a 到顶点 g 的最短路径（即 a→b→d→e→f→g），值为 13。

表 8.6 Bellman-Ford 最短路径算法计算出的 $dist_k$[]值

k	$dist_k[a \ldots g]$						
	a	b	c	d	e	f	g
1	0	5	10	∞	∞	∞	∞
2	0	5	10	15	∞	25	18
3	0	5	10	15	17	25	16
4	0	5	10	15	17	22	16
5	0	5	10	15	17	22	13
6	0	5	10	15	17	22	13

表8.7所示为Bellman-Ford最短路径算法。

表 8.7 Bellman-Ford 最短路径算法

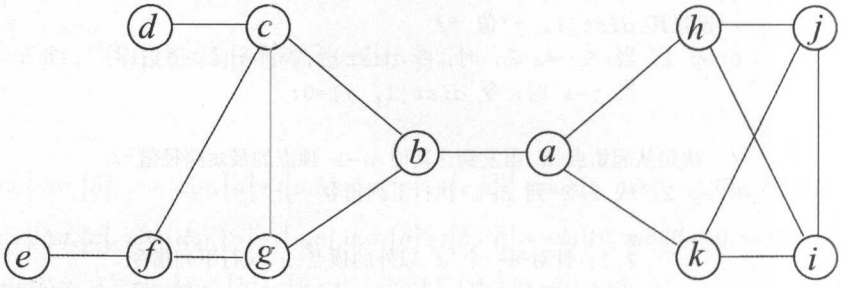
输入	一个图 G =(V ={1, 2, 3, ⋯, n }, E)，起始点为a，每条连线 e =(i, j)上的权重为t[i, j]（注意，此处权重可为负值，且当 i 不连接 j 时，设 t[i, j] = ∞，1≤i , j≤n)
输出	a到其他每一个顶点的最短路径长度，即 dist[n-1, j]存储从a到j的最短路径的长度

步骤	Algorithm Bellman_Ford_Shortest_Path (*a*, *t*[], *dist*[], *n*)
	{
	/*设置其 *dist*[1, *j*]值 */
	Step 1: 当1≤*j*≠*a*≤*n* 时，令 *dist*[1, *j*]=t[*a*, *j*];
	当 *j*=*a* 时，令 *dist*[1, *j*]=0;
	
	/* 决定从起始点 *a* 出发到达其他 *n*-1 顶点的最短路径值*/
	Step 2: 从 *i*=2 到 *n*-1 执行下列指令
	{
	2.1: 针对每一个 *a* 以外的顶点 *u* 执行下列指令
	{
	/* 注意此处的 *j* 需要考虑所有可直接连接到*u*的顶点*/
	计算dist[*i*, *u*] =min{*dist*[*i*-1, *u*] , min{*dist*[*i*-1, *j*] +*t*[*j*, *u*] }};
	}
	}
	}

8.5 双连通分支

最后一个例子是双连通分支（bi-connected component）问题，如表8.8所示。

表 8.8 双连通分支问题

问题	在一个网络中，当一个节点的损坏导致剩余网络产生不连通现象时，此节点称为关节点(articulation point)。小王想要从现有的网络中找出一个尽可能大，但不存在关节点的子网，以便进行网络攻击测试。 请设计一个算法，列出一个网络中所有符合的子网
输入	一个图 $G=(V, E)$代表现有的网络。每个点代表网络上的节点，而节点间的连线代表节点间是连通的 
输出	所有不含关节点的最大子网 {a, b} {b, c, f, g} {c, d} {e, f} {a, h, i, j, k}

以表8.8中输入的网络为例，顶点 a 的损坏会导致剩余网络不连通。明确地说，剩余网络为两个连通分支，如图8.16(a)和(b)所示。

因此，顶点 a 为一关节点，其他关节点有顶点 b、c、f。一个连通图（connected graph）不含有关节点，就被称为双连通图（bi-connected graph）。显然，双连通图拥有较强的容错能力。

图8.16(b)是一个双连通图，但是图8.16(a)不是双连通图。一个图 G的最大双连通子图（subgraph）H被称为双连通分支（bi-connected component），代表在图G中找不到比H更大的子图，包含H且是双连通的。表8.8中的输出{a, b}、{b, c, f, g}、{c, d}、{e, f}、{a, h, i, j, k}都是输入图的双连通分支。简单地说，双连通分支问题就是为一个图找到所有的双连通分支。

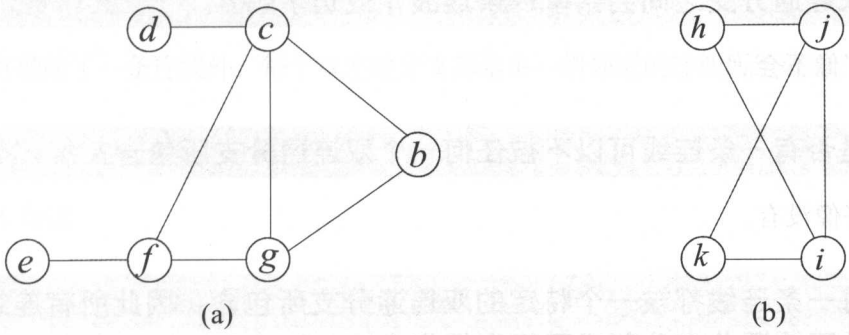


图8.16 顶点 a 的损坏导致剩余网络不连通

8.5.1 双连通分支的性质

想设计一个算法找到一个输入图的所有双连通分支似乎不容易。首先，将此范例的每个双连通分支用一个圆圈绘出，以方便找出其特点并进行讨论，如图8.17所示。

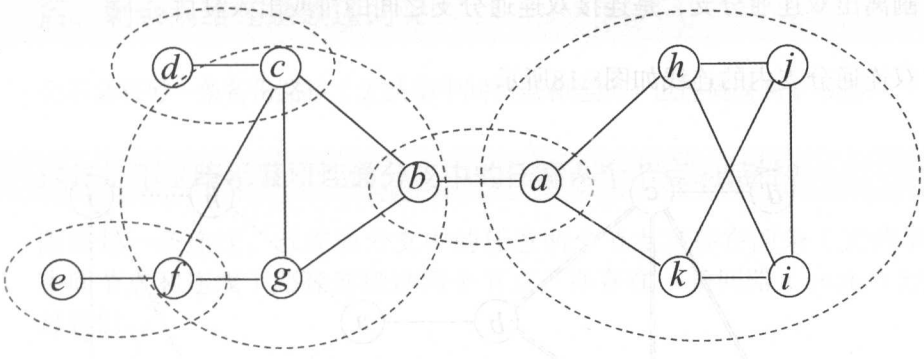


图8.17 每一个圈内的子图代表一个双连通分支

双连通分支之间有什么关系?

只共享一个顶点。

这些共享的顶点有什么性质?

都是关节点。

双连通分支之间会共享一条连线（或边）吗？

好像不会。

是否有一条连线可以不被任何一个双连通分支所包含？

好像没有。

每一条连线都被一个特定的双连通分支所包含，因此所有连线被双连通分支分割成不相交的集合？

对极了。

所有顶点被双连通分支分割成不相交的集合吗？

不！因为有共享的现象。

这些共享的顶点有什么特质？

割离出双连通分支，是连接双连通分支之间的桥或出入渡口。

双连通分支内的连线如图8.18所示。

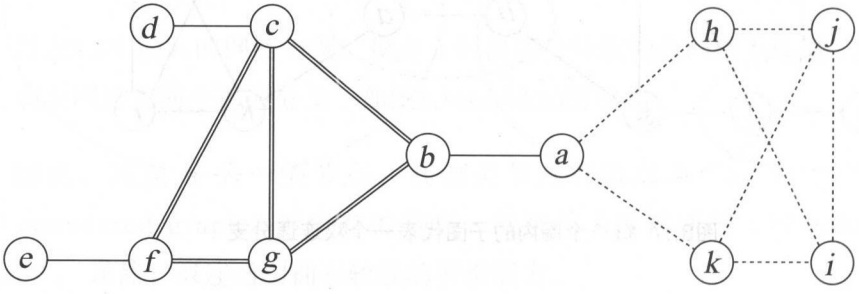


图8.18 每个双连通分支内的连线集合隐藏着一个神秘的关系

哪些连线会落在同一个双连通分支？

不知道。

试着观察同一个双连通分支中连线之间的特性。

好像除了一条连线外，每个双连通分支都是由一组邻近的连线所组成的。

这一组连线集合中藏着怎样的关系？

不知道。

回到定义，双连通分支的定义是什么？

本身不存在关节点的最大连通子图。

此网络不存在关节点，有什么特性？

当此网络中任意一个节点损坏时，剩余网络的任意两个节点还是连通的，也就是任意一个节点的失效无法破坏任意两个节点的连通。

因为原来网络是连通的，所以会有一条路径连接任意两个节点。但是为什么当任意一个节点损坏时，导致这个路径断裂后，剩余网络还是连通的呢？

会不会有另一条备份路径（无共享中间顶点和连线）也连接这两个节点？

这样是不是暗示着双连通分支中的任意两个节点的特性？

除非是一条连线，双连通分支中的任意两个节点将存在两条（无共享中间节点和连线）路径连接这两个节点，即存在一条回路（cycle）经过它们。”

下面列出上述有关双连通分支的特性。

（1）两个双连通分支最多共享一个节点，此节点为整个网络的关节点。

（2）每一条连线被唯一的双连通分支所包含。

（3）关节点将双连通分支隔离并串接成整个网络。

（4）每个双连通分支中的任意两条连线（或边）会被一个不重复节点的

回路所经过。借助上面双连通分支的特性，我们将有机会找出一个输入图的所有双连通分支。

8.5.2 设计一个简单的算法

想要设计一个双连通分支算法，必须将所有图中的连线分割成不同的双连通分支。

目前已知，当两条连线被一个不重复节点的回路经过时，这两条连线属于同一个双连通分支。同理可得，所有在此回路中的连线都属于同一个双连通分支，如图 8.19 中的连线 [c, f]、[f, g]、[g, c] 属于同一个双连通分支。

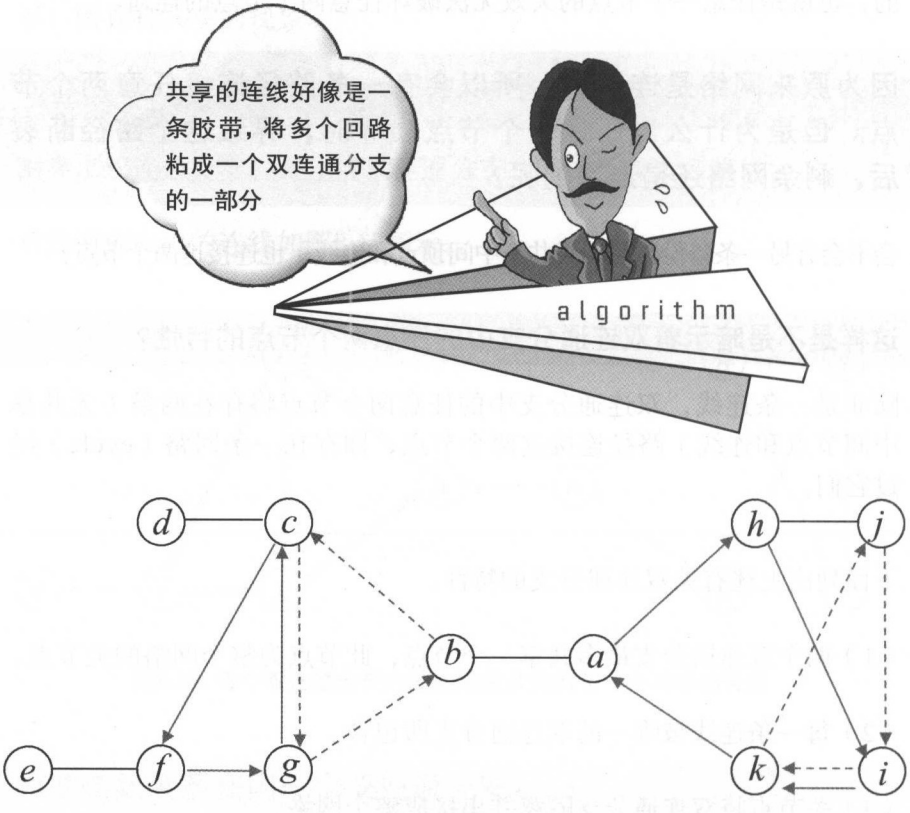


图8.19 共享连线将多个回路粘成一个双连通分支的一部分

更有甚者，当有两个回路共享一些连线时，这两个回路中的所有连线需与此共享连线属于同一个双连通分支。如图8.19中的回路，*c*—*f*—*g*—*c* 和回路 *b*—*c*—*g*—*b* 共享连线[*c*, *g*]，故[*c*, *f*]、[*f*, *g*]、[*g*, *c*]、[*b*, *c*]、[*g*, *b*]应属于同一个双连通分支；又如回路：*a*—*h*—*i*—*k*—*a* 和回路 *i*—*k*—*j*—*i* 共享连线[*i*, *k*]，故[*a*, *h*]、[*h*, *i*]、[*i*, *k*]、[*k*, *a*]、[*k*, *j*]、[*j*, *i*]也应属于同一个双连通分支。

为了找出所有双连通分支，目前最直接的方式是将所有的回路找到，并将一个回路和与其有共享连线的回路中的连线并集成一个双连通分支的一部分。但是，所有回路的数目有可能过大，导致此方法的效率不高。

另一种方法是先找出所有关节点，因为关节点将整个网络隔离成多个双连通分支。

撒找出所有关节点，可以试着将某一节点删除，并测试剩下的图是否依然连通，以此判断出所有关节点，但此法的效率显然不高。

8.5.3 深度优先搜索生成树的特性

另一种较有效率的方法是利用深度优先搜索法找出所有关节点和双连通分支。

此法需要利用深度优先搜索（Depth First Search，DFS）生成树的以下特性：

（1）DFS 生成树加上任意一条不在此树上的连线称为反向连线（back edge，或称为反向边），将形成唯一的回路。

（2）任意一条反向连线，将直接连接此树中的祖先及其后代，如图8.20虚线所示。

（3）图中的所有连线只有两种：DFS 生成树上的连线和连接 DFS 生成树中祖先及其后代的反向连线。我们所设计的算法需要将此两种连线分类成不同的双连通分支。

首先，讨论因为加入一条反向连线所牵连出的双连通分支。注意，在图8.20中，加入反向连线 [k, a] 所形成的回路 a—h—i—j—k—a，加入反向连线 [k, i] 所形成的回路 i—j—k—i，以及加入反向连线 [j, h] 所形成的回路 h—i—j—h，因为共享（或共用）DFS生成树上的连线，导致这3条回路属于（粘接成）同一个双连通分支。

相反地，如果有 DFS 生成树上的一条连线不落在任何这样的回路中，那么它只好自己组成一个双连通分支，如图8.20中的 [a, b]、[c, d] 和 [e, f]。以上特性有助于找到每条双连通分支中的连线。

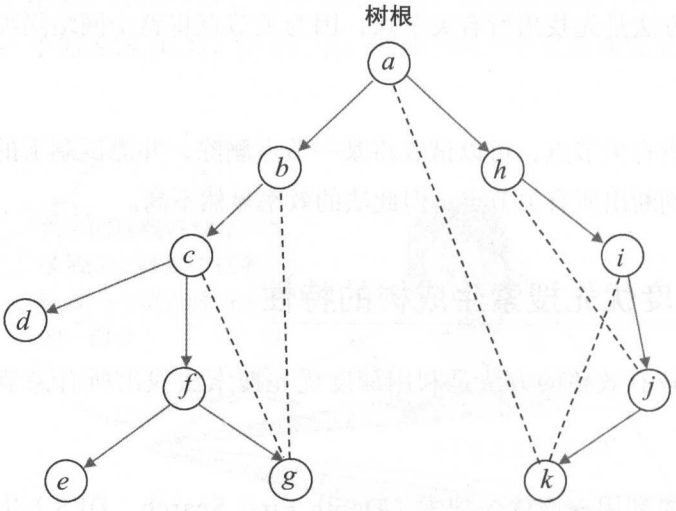


图8.20 深度优先搜索的过程形成一个 DFS 生成树（实线）

利用 DFS 生成树似乎有机会简单地找出回路，并且判断两个回路之间是否存有共享连线（或共享的边），进而找到双连通分支。

8.5.4 利用深度优先搜索法找出关节点

本节将利用 DFS 生成树上的反向连线所形成的回路界定双连通分支和关节点的位置。注意，关节点位居于双连通分支的边界，若能有效地辨认出图中的关节点，将有助于设计出一个双连通分支算法。

首先进行关节点的简单性质说明。在 DFS 生成树上，一个树根（root）若是关节点，则最少有两个以上的儿子（child）（因为去掉树根后，两个儿子所属的图将不会有路径相连通）；反之亦然。例如，图8.20中的树根 a 拥有两个儿子，故为关节点。

当节点 u 不是树根时，下列关节点的两个特性是存在的：

（1）如果节点 u 存在某一个儿子 w，利用 w 及其后代和一条反向连线无法连通到 u 的祖先，那么节点 u 是一个关节点。

原因是在一个连通图中，除了 DFS 生成树上的连线，剩下的就是反向连线了。删除节点 u 后，w 和 u 的祖先将无法通过反向连线连通。因此，节点 u 是一个关节点。

例如，图8.20中的节点 b 有一个儿子 c，利用 c 及其后代和一条反向连线无法连通到 b 的祖先（最多只能连到 b）。若去掉 b，则 c 将无法连通到 b 的祖先。因此，节点 b 是关节点。注意，节点 c(f) 也是关节点，因为此节点都存在一个儿子 d(e)，利用此儿子及其后代和一条反向连线不可以连通到 c(f) 的祖先。

（2）相反地，如果从节点 u 的每一个儿子 w，利用 w 及其后代和一条反向连线都可以连通到 u 的祖先，那么节点 u 不是一个关节点”

原因是删除节点 u 后，其所有儿子及其后代和树上的其他节点依然连通着。因此，节点 u 不是一个关节点。

例如，图8.20中的节点 i 只有一个儿子 j，利用 j 及其后代和一条反向连线可以连通到 i 的祖先 h 或 a，故节点 i 不是关节点。

若想利用上述特性判断某节点是否为关节点，则需要知道利用此节点及其后代和一条反向连线可以连通到的祖先为哪一个节点。因此，我们将为每个节点进行编号，以方便辨识。

首先，按照深度优先搜索遍历的顺序将节点从小到大进行编号（见图8.21），此编号被称为深度优先编号（Depth First Number，DFN），如 DFN

(a) = 1，DFN (h) = 8，而 DFN (i) = 9。在深度优先搜索遍历的过程中，因为祖先总是比后代先被“拜访”到（即遍历到），所以祖先的 DFN 编号永远比其后代小。

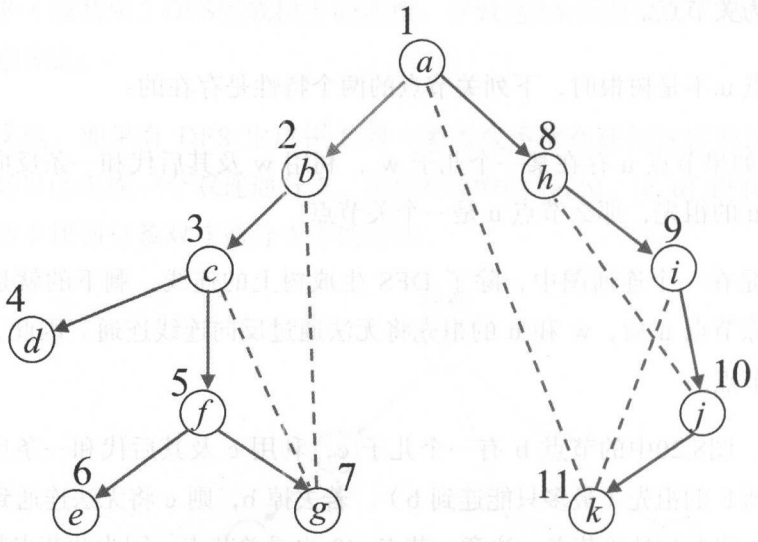


图8.21 在深度优先编号中，祖先的编号永远比其后代小

利用此特性，当执行深度优先搜索法遍历到反向连线时，记录此连线连通到最老祖先的 DFN 编号，并且将其返回给需要的节点，将可以有效地借助上述两个特性判断关节点。明确的做法是针对每一个节点 u定义一个函数：

$$L(u)=min\{\ DFN(u),\ min\{L(w)\text{是}w\ \text{是}\ u\ \text{的儿子}\},\ min\{DFN(x)\mid(u,x)\text{是一条反向连线}\}\}$$

L(u) 代表利用节点 u 及其后代所形成的路径，以及最多一条反向连线可以连通到的最老祖先的 DFN 编号。

如先前讨论的，若节点 u不是树根，则节点 u 是一个关节点，当且仅当节点u存在某一个儿子w时，利用w及其后代和一条方向连线无法连通到u的祖先。

上述条件若使用 DFN 和函数 L 表示，则节点 u 是一个关节点，当且仅当节点u有一个儿子w时，其L(w)≥DFN(u)。

例如，在图8.22中，节点 b 有一个儿子 c，L(c) = 2 ≥ DFN(b) = 2，故节点 b 是一个关节点。又例如，节点 c 有一个儿子 d，L(d) = 4≥DFN(c) = 3，故节点 c 是一个关节点。反之，节点 i 只有一个儿子 j，L(j) = 1 < DFN(i) = 9，故节点 i 不是一个关节点。计算函数 L 的值可在进行深度优先搜索遍历时利用后序法（postorder）得出。有关计算 DFN 和 函数 L 值的算法，可参考表8.9。

表 8.9 计算 DFN 和函数 L 值的算法

输入	图 G =(V ={1, 2, 3, ⋯, n }, E)，起始节点为u，且节点v是在深度优先搜索生成树中u的父节点（如果存在）。设初始值 DFN[]=0 且 num=1
输出	每个节点在深度优先搜索生成树中的DFN和函数L值

步骤	Algorithm Compute_DFN_L (*u*, *v*)
	{
	/*设置其起始节点的值*/
	Step 1: *DFN*[*u*]=num; *L*[*u*]=num; num =num+1
	Step 2: 针对每一个节点u的相邻节点w，执行下列指令
	{
	2.1: if *DFN*[*w*]=0 then　　/* 节点w未被遍历过 */
	{　　/* 继续执行深度优先搜索 */
	调用 Compute_DFN_L(*w*, *u*);
	L[*u*]=*min*(*L*[*u*], *L*[*w*]);
	}
	else　　/* 若点 w 被遍历过，则(*u*, *w*)是一条反向连线，执行L[u]计算*/
	{if *w*≠*v* then *L*[*u*]=*min*(*L*[*u*], *DFN*[*w*]); }
	}
	}

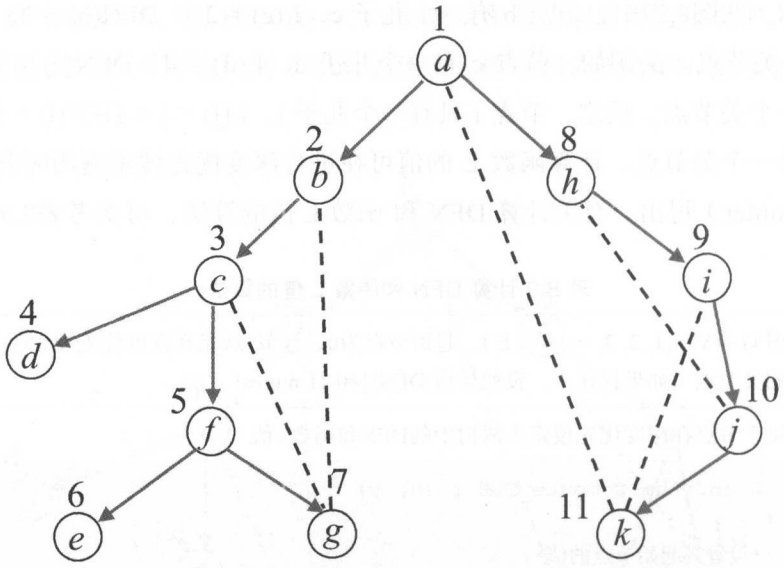


图8.22 计算每个节点函数的 L 值：L(a)=1, L(b)=2, L(c)=2, L(d)=4, L(e)=6, L(f)=2, L(g)=2, L(h)=1, L(i)=1, L(j)=1, L(k)=1

计算 DFN 和函数 L 值的算法显然可以在 O(n+e) 的时间内完成（此处的 n 为图的节点个数，而 e 为连线的条数），因此所有关节点都可在 O(n+e)的时间内找到。

8.5.5 利用深度优先搜索法找出所有双连通分支

至此，我们知道使用函数 L() 和 DFN() 可以辨认出关节点。接下来找出所有双连通分支。

一个双连通分支内的连线有两个特色：(1) 被关节点所分割；(2) 在深度优先搜索的过程中，除非被其他双连通分支的连线所占据，同一个双连通分支内的连线是连续出现的。

例如，在图8.23中，按照深度优先搜索法对图中连线的遍历顺序进行编号，则编号{9、10、11、12、13、14、15} 等连续7条连线属于同一个双连通分支；但是编号 {2、4、6、7、8} 不连续，却也属于同一个双连通分支，原因是其中夹着两个双连通分支，即{3}和{5}。若排除{3}和{5}，则编号 {2、4、

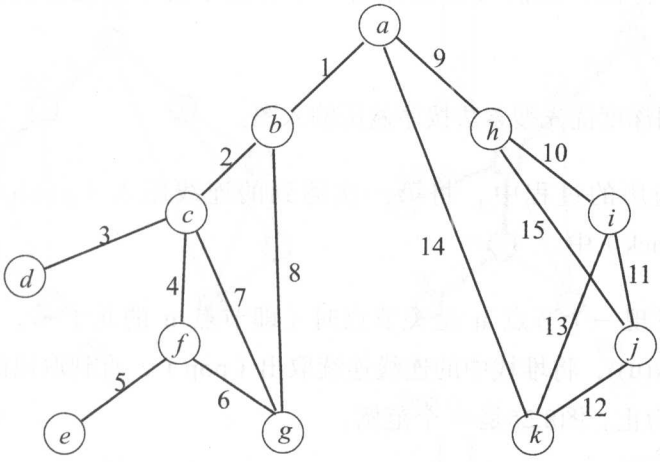


图8.23 按照深度优先搜索法对所有连线进行编号

6、7、8} 的连线可视为连续的。

这个特性其实是：若将每一个双连通分支视为一个节点，则整个连通图就是一棵树（tree），如图8.24所示。

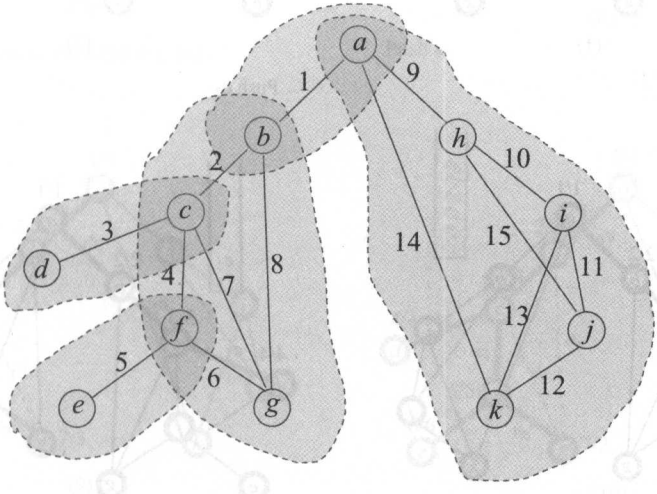


图8.24 每个连通分支可视为树中的一个节点

按照上述特性，我们可以设计出一个有效率的双连通分支算法，概念如下：

（1）使用深度优先搜索法按序遍历输入图。

（2）在遍历的过程中，将第一次遇到的连线压入（push）一个堆栈（stack）中。

（3）当发现一个节点 u 是关节点时（即节点 u 的儿子 w，其 L(w) ≥ DFN(u)），将堆栈中的连线连续取出（pop），直到取出的连线为 (u, w) 为止。图8.25是一个范例。

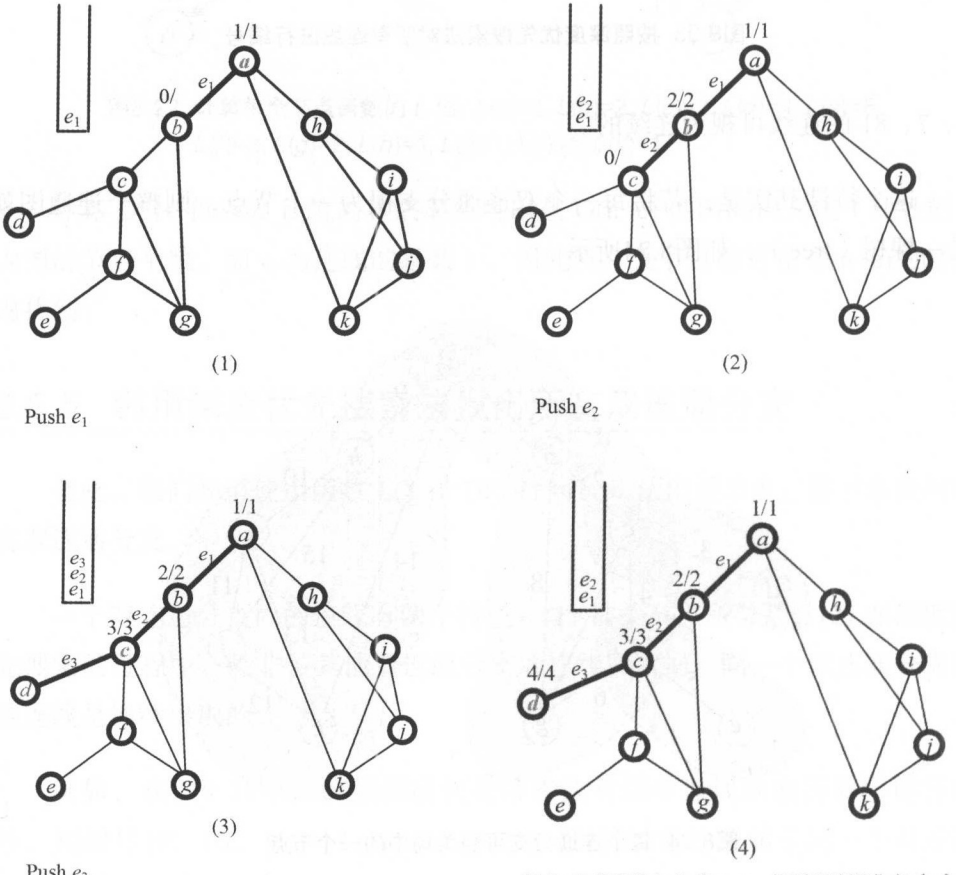
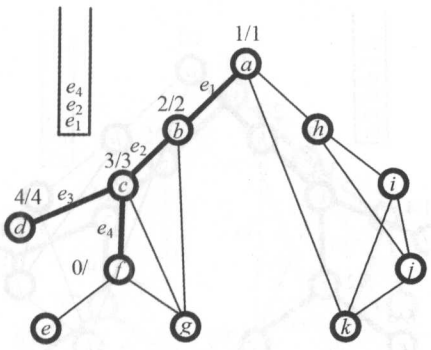
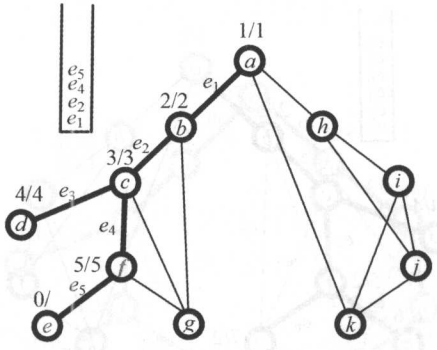


图8.25 双连通分支算法的范例，每个节点 u 旁的两个整数为 DFN(u)/L(u)



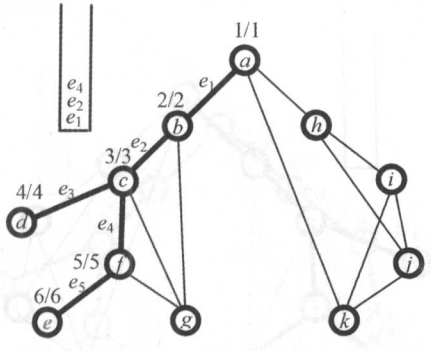
(5)

Push e_4



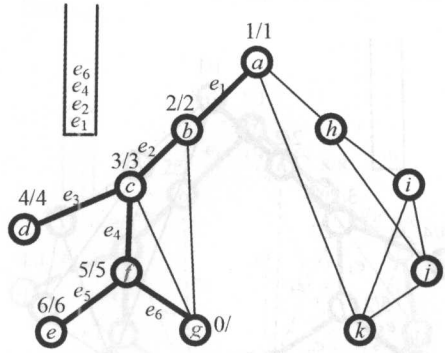
(6)

Push e_5



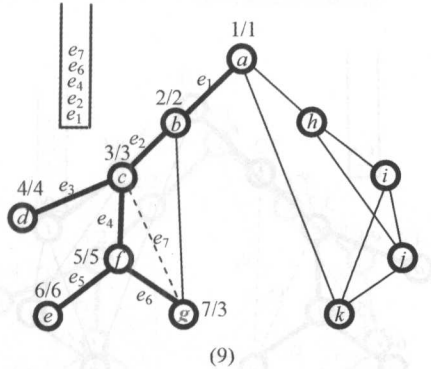
(7)

$L(e)=6\geq DFN(f)=5$, Pop e_5, 找到双连通分支: $\{e_5\}$



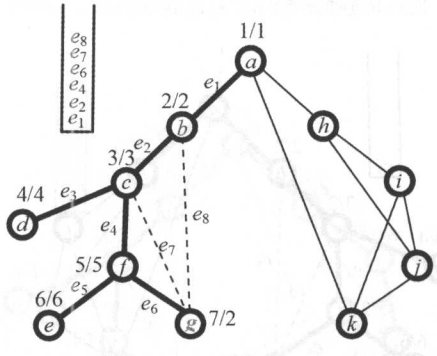
(8)

Push e_6



(9)

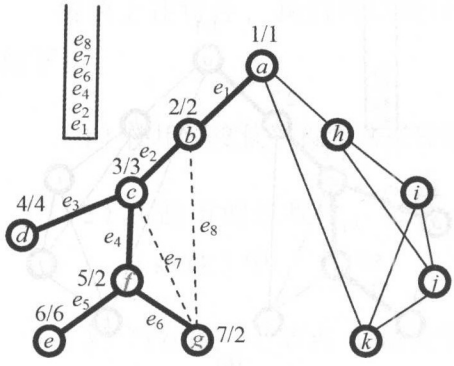
Push e_7, $L(g)=min\{L(g)=7, DFN[c]=3\}=3$



(10)

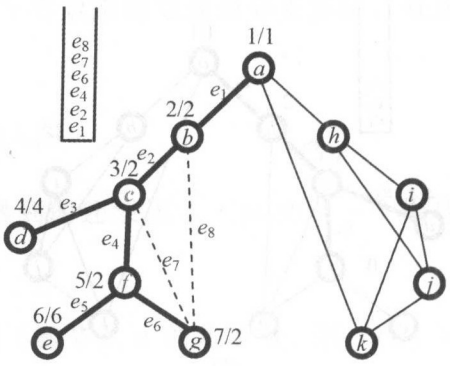
Push e_8, $L(g)=min\{L(g)=3, DFN[b]=2\}=2$

图8.25 双连通分支算法的范例，每个节点 u 旁的两个整数为 DFN(u)/L(u)（续）



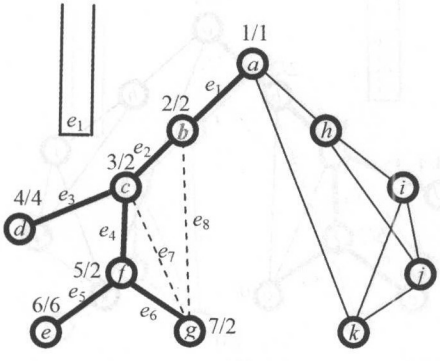
(11)

$L(f)=min\{L(f)=5, L(g)=2\}=2$



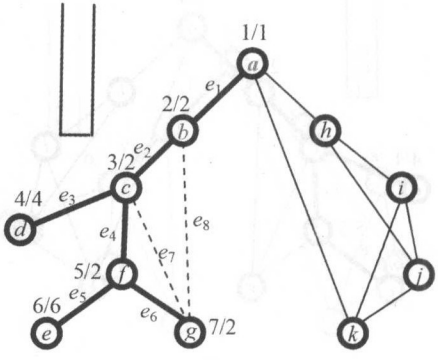
(12)

$L(c)=min\{L(c)=3, L(f)=2\}=2$



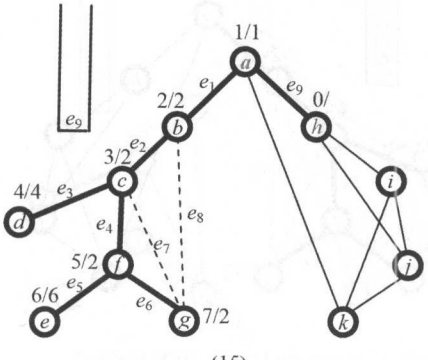
(13)

$L(c)=2\geq DFN(b)=2$, 连续 Pop e_8, e_7, e_6, e_4, e_2,
找到双连通分支: $\{e_2, e_4, e_6, e_7, e_8\}$



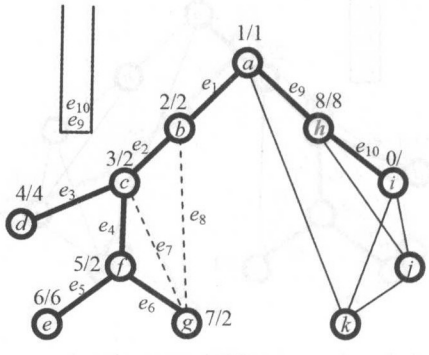
(14)

$L(b)=2\geq DFN(a)=1$, Pop e_1,
找到双连通分支: $\{e_1\}$



(15)

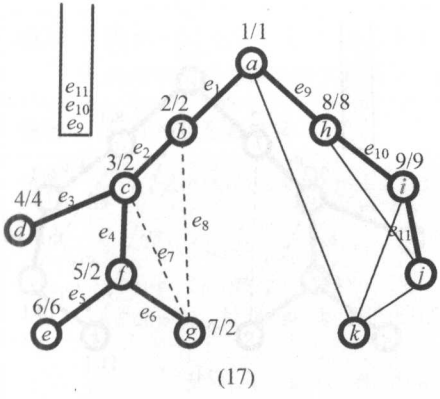
Push e_9



(16)

Push e_{10}

图8.25 双连通分支算法的范例，每个节点 u 旁的两个整数为 DFN(u)/L(u)（续）



Push e_{11}

Push e_{12}

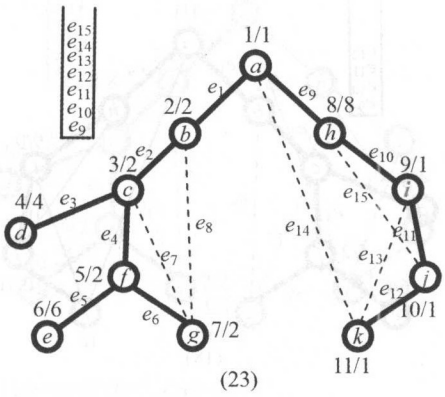
Push e_{13}, $L(k)=min\{L(k)=11, DFN[i]=9\}=9$

Push e_{14}, $L(k)=min\{L(k)=9, DFN[a]=1\}=1$

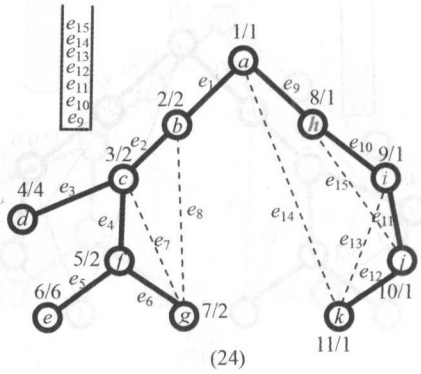
$L(j)=min\{L(j)=10, L(k)=1\}=1$

Push e_{15}, $L(j)=min\{L(j)=1, DFN[h]=8\}=1$

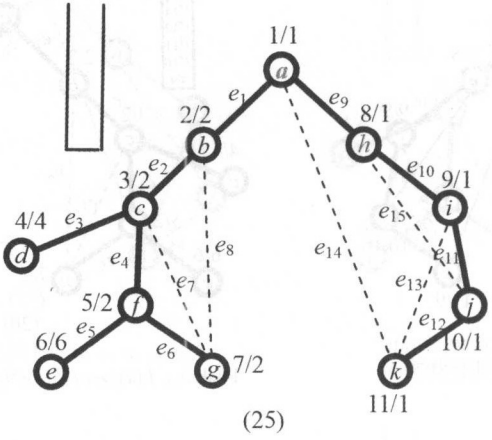
图8.25 双连通分支算法的范例，每个节点 u 旁的两个整数为 DFN(u)/L(u)（续）



$L(i)=min\{L(i)=9, L(j)=1\}=1$



$L(h)=min\{L(h)=8, L(i)=1\}=1$



$L(h)=1 \geq DFN(a)=1$，连续 Pop e_{15}, e_{14}, e_{13}, e_{12}, e_{11}, e_{10}, 找到双连通分支：$\{e_9, e_{10}, e_{11}, e_{12}, e_{13}, e_{14}, e_{15}\}$

图8.25　双连通分支算法的范例，每个节点 u 旁的两个整数为 DFN(u)/L(u)（续）

根据以上讨论，我们可以修改计算 DFN 和函数 L 值的算法（见表 8.9），得到双连通分支算法（见表8.10）。注意，此双连通分支算法的时间复杂度仍为 O(n+e)（此处的 n 为图的节点个数，而 e 为连线的条数）。

表 8.10 双连通分支算法

输入	图 G =(V ={1, 2, 3, …, n }, E)，起始节点为 u，且节点v为节点u在深度优先搜索生成树中的父节点（如果存在）。设初始值 DFN[]=0 且 num =1
输出	所有的双连通分支

步骤	Algorithm bi-connected_component (u, v)
	{
	/*设置其起始节点的值*/
	Step 1: DFN[u]=num; L[u]=num; num =num+1
	Step 2: 针对每一个节点u的相邻节点w，执行下列指令
	{
	2.1: if w≠v 且 DFN[w]<DFN(u) then 将连线(u, w) push 到堆栈 S 中
	2.2: if DFN[w]=0 then 调用 bi-connected_component (w, u);
	/*节点w未被遍历过*/
	{
	L[u]=min (L[u], L[w]);
	if L[w]≥DFN[u] then 输出 "以下是一个新的双连通分支";
	/*发现节点u是一个关节点，输出一个双连通分支*/
	{
	重复从堆栈S中pop一条连线（并将此连线输出），直到此连线为(u, w)或(w, u)为止;
	}
	}
	else /*节点w被遍历过*/
	{if w≠v then L[u]=min (L[u], DFN[w]); }
	}
	}

8.6 图算法的技巧

怎样的问题可以使用图来解决呢?

不知道。

什么是图?

几个节点使用几条连线连接起来。

如果每个节点代表一个体，那么连接两个节点的一条连线代表两个体之间的什么？

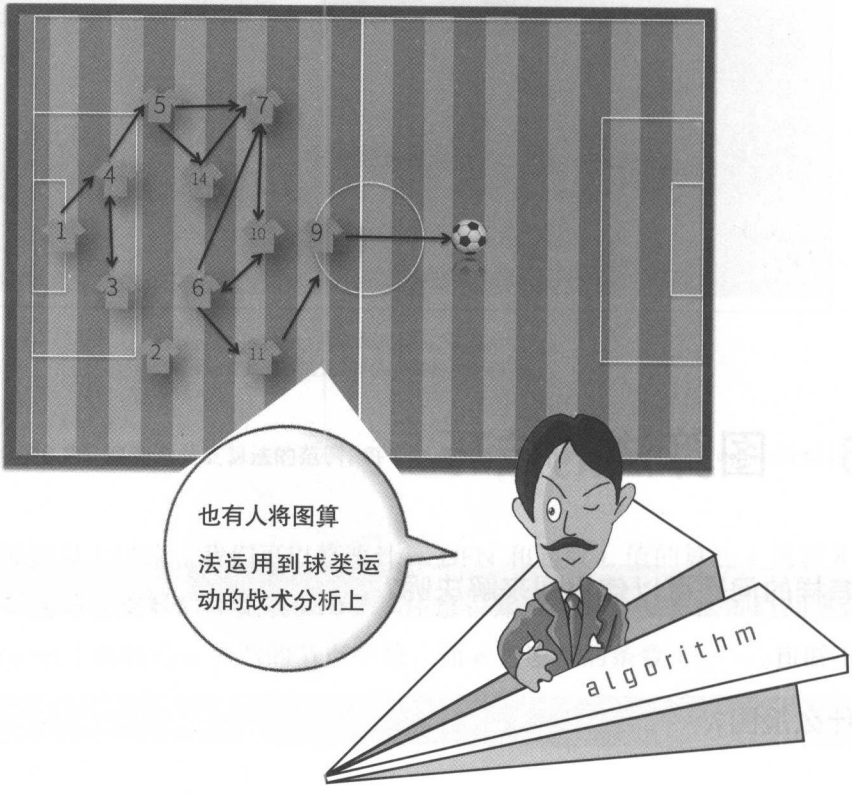
会不会是这两个体之间的一种关系？

怎样的问题可以使用图来解决呢？

当一个问题牵扯到若干体之间的某种关系时，可能可以使用图来解决。

图算法的技巧是将一个问题表达成一个图之后，在此图上设计算法，以找到问题的解。

一般而言，当一个问题探讨有关离散个体之间的二元关系（binary relation）时，此问题常可以被抽象成图上的问题。若设计出的算法能充分地使用到这个图的特性，则将更有效率地解决这个问题。



学习效果评测

1. 图 G = (V, E) 是一个无回路（loop-free）且连通的无向图。一个在集合 V 中的顶点 v 被称为关节点，则图 G 除去 v 后，是不连接的。编写一个程序，找出所有输入图的关节点。

输入

```
6                (顶点的个数，并且以1，2，3 …编号)
8                (连线的条数)
1 2              (以下为连线的数据)
2 3
2 4
2 5
3 4
3 5
4 5
5 6
```

输出

```
2 5              (所有关节点)
```

2. 一个村子有 N 位村民，已经知道哪些人彼此是朋友。俗话说：四海之内皆兄弟，也就是如果 A 和 B 是朋友，且 B 和 C 是朋友，则 A 和 C 也是朋友。编写一个程序，计算此村内共有多少组村民彼此拥有朋友关系。

输入

```
7                (村民的人数 N，并且以1，2，3 …编号)
7                (朋友关系的个数)
1 5              (以下为关系的数据)
2 3
2 4
4 3
1 6
5 1
7 1
```

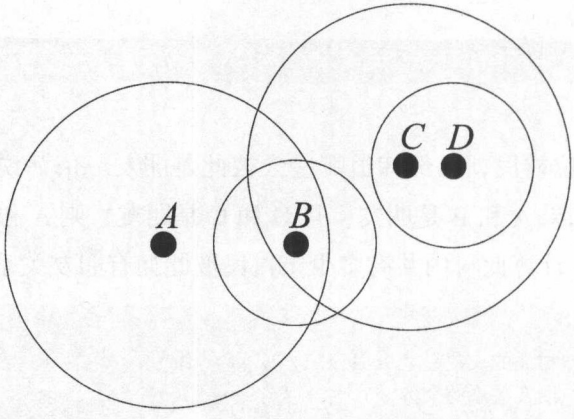
输出

```
2                (拥有朋友关系的群体数目)
```

3. 收集器（sinks）：无线传感器网络（wireless sensor networks）是由大量、小尺寸、低成本的传感器所组成的。每个传感器可侦测各种信号（包含温度、湿度、照度或声音），并将收集到的信号通过邻近的传感器传送到网络上的收集器中。在一个无限传感器网络中，选择一个特定的传感器当作收集器是一个基本的问题。

   当两个传感器的距离小于或等于 R_A 单位长度时，传感器 A 可以直接传数据包给传感器 B 。我们称传感器 A 的通信半径是 RA 单位长度。明确地说，当 (x_1, y_1) 和 (x_2, y_2) 分别是 A 和 B 的坐标时，如果 $(x_1 - x_2)^2 + (y_1 - y_2)^2 \leqslant R_A^2$，那么传感器 A 可以直接传数据包给传感器 B 。注意，因为每个传感器的通信半径可能不同，当传感器 A 可以直接传数据包给传感器 B 时，传感器 B 不必直接传数据包给传感器 A 。

   假设每个传感器的通信范围是一个圆圈，其中心点就是传感器的位置。



   在上图中有4个传感器 A 、B 、C 、D 部署在平面。假定传感器 A 、C 的半径是 4 个单位长度（即$R_A = R_C = 4$），而传感器 B 、D 的半径是两个单位长度（即 $R_B = R_D = 2$）。因为 A (C) 圆圈包含 B 的中心点，所以A (C) 可以直接和 B 通信。但是 B 不能直接和 A (C) 通信，因为 B 的通信范围并没有包含 A (C) 的中心点。同样地，D 可以直接和 C 通信，但是 D 不能直接传送数据包给 B 或 A 。在上图中，明显地只有传感器 B 可以担任收集器的工作。

编写一个程序，计算一个输入的传感器网络中最多有几个传感器可当作收集器。

输入

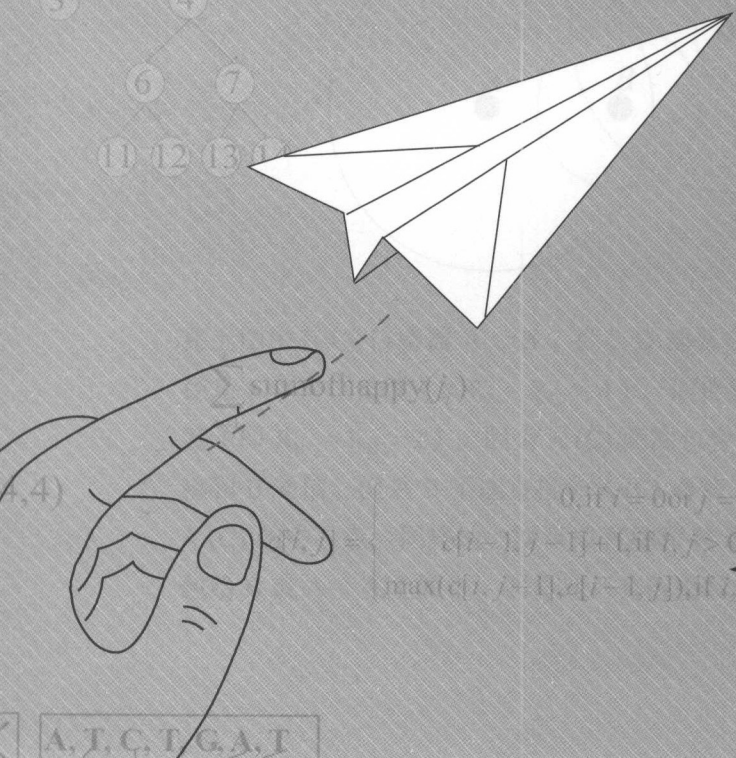
```
4                    (传感器的总数)
1 6 4                (以下是传感器的x、y坐标和半径)
4 6 4
7 8 4
8 8 4
```

输出

```
4                    (收集器的个数)
```

计算几何

章节大纲



9.1　何谓计算几何

什么是计算几何（computational geometry)?

简而言之，就是输入几何上的对象，并从这些对象中寻找解答的技巧。

当在设计集成电路时，减少整体电路所占用的面积有助于降低其制造成本。降低占用的面积时，需要考虑如何将不同大小的长方形挤入一个给定的空间中，如图9.1所示。另一个例子是，思考部署一个无线传感器以监控所有目标物。传感器的监控范围可用一个圆表示，而每一个目标物可用一个点表示。如何找出一个最小的圆来覆盖平面上所有的点，就是将此无线传感器的部署转换成几何计算的问题。

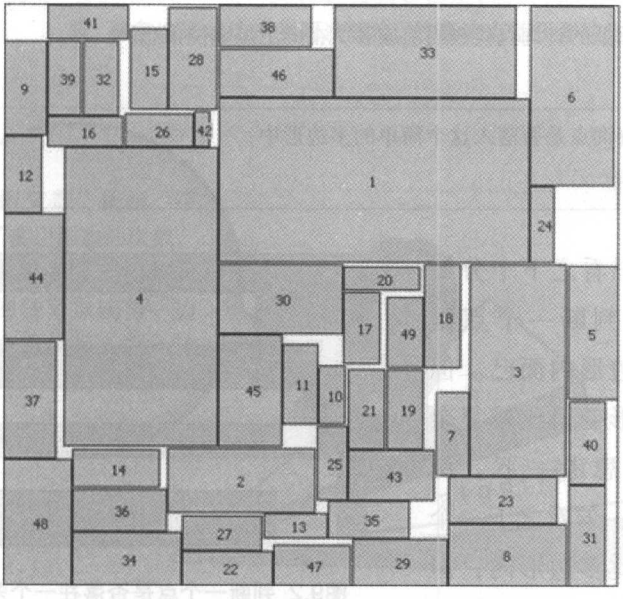


图9.1　减少整体电路的制造成本，需要考虑如何将不同大小的长方形挤入给定的空间中

9.2 多边形中的点

第一个例子是判断一个点是否在一个简单多边形内，如表9.1所示。

表 9.1 判断一个点是否在一个简单多边形内

问题	有一位外国旅客来到上海市旅游，他想利用手中的卫星定位信息查询目前所在的区域（如是否在静安区）。 请替他设计一个算法解决此问题
输入	在平面上，由一连串相邻的直线或横线所组成的简单多边形代表一个用户感兴趣区域的地图。 (105, 18)-(129, 18)-(129, 2)-(109, 2)-(109, 5)-(127, 5)-(127, 16)-(110, 16)-(110, 13)-(124, 13)-(124, 10)-(108, 10)-(108, 8)-(122, 8)-(122, 6)-(106, 6)-(106, 0)-(105, 0)-(105, 18) 一个平面上的查询点代表当前旅客所在的卫星定位信息： (123, 9)
输出	判断此查询点是否落入这个简单的多边形中： 否

这个问题乍看之下十分简单，因为只是判断一个点是否落在一个多边形内而已。但是当这个多边形变得比较复杂时，就需要多思考一下。例如，在图9.2中，乍看之下一个黑点好像落入此多边形内，但是仔细一瞧才发觉其实落在了多边形外。

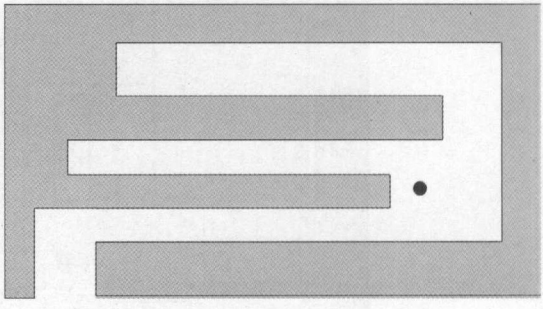


图9.2 判断一个点是否落在一个多边形中

下面用一个简单方法进行判断。我们可以任选多边形外的一点，将此点连接到查询点，以形成一个线段。接下来，计算此线段和多边形的边所形成的相交数。若相交数为奇数，则此查询点在多边形内；若相交数为偶数，则此查询点不在此多边形内。

例如，在图9.3中，此线段和多边形的边产生两次相交，故查询点落于多边形外。

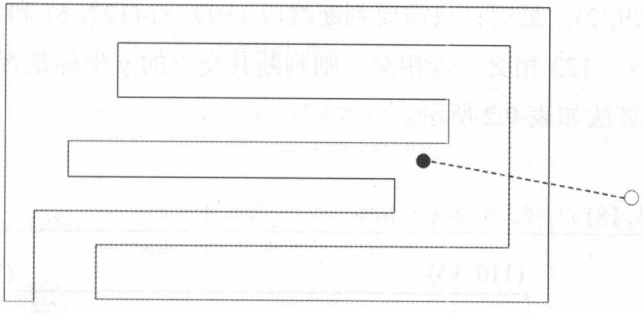


图9.3　多边形外的一点（白点）连接到查询点（黑点）形成一个线段，和多边形的边产生两次相交，故查询点落于多边形外

接下来，考虑此方法的细节。当输入的多边形使用一连串二维坐标记录相邻边的转角点时，我们该如何计算所需的相交数呢？

乍看之下，好像需要一些额外的计算才能完成。但是，若能适当选择多边形外的点，将有助于进一步简化这种判断。

例如，在图9.4中，当我们选择一点，使得此点连接到查询点(123, 9)的线段与 y 轴平行时，此多边形会与此线段相交，只有两条边，即 (109, 5)-(127, 5) 和 (109, 2)-(129, 2)。最后，只需要判断线段 (109, 5)-(127, 5) 和 (109, 2)-(129, 2)是否和直线 x = 123 相交。若相交，则判断其交点的 y 坐标是否小于或等于 9 即可。详细的算法如表 9.2 所示。

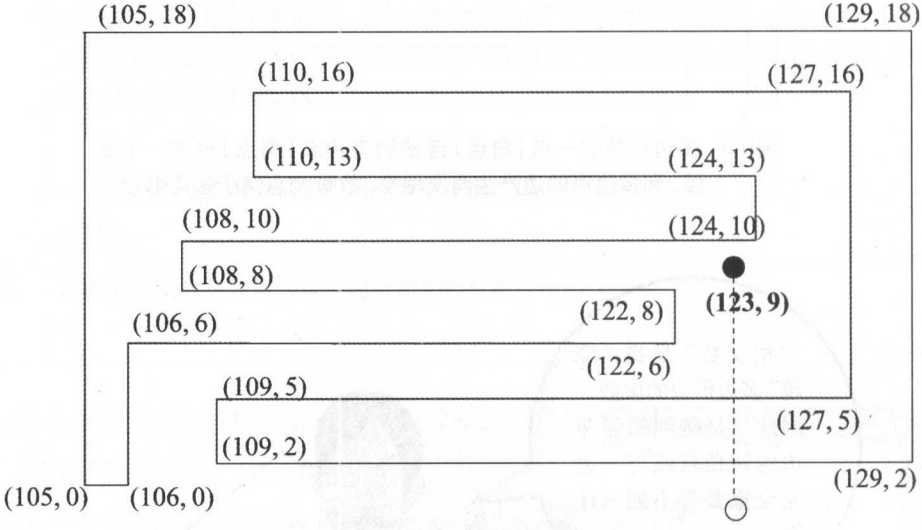


图9.4 使用一连串的二维坐标代表输入的多边形，此时查询点（黑点）的坐标为（123, 9）

表 9.2 判断点在多边形中的算法

输入	平面上的一个简单多边形 P，其中 n 个点为 $p_1=(x_1, y_1)$, $p_2=(x_2, y_2)$, …, $p_n=(x_n, y_n)$。其中 n-1 条边 e_i 是由 $P_i=(x_i, y_i)$到 $P_{i+1}=(x_{i+1}, y_{i+1})$的直线或横线所组成的($i$=1, …, n-1)。最后的一条边是由 $p_n=(x_n, y_n)$到 $p_1=(x_1, y_1)$的线段组成的。 一个平面上的查询点 $q=(x_0, y_0)$
输出	判断此查询点q是否落入这个简单的多边形 P中

步骤	Algorithm point_in_polygon (p,q)
	{
	Step 1: number:=0; /*相交数初值为 0*/
	Step 2: for 所有多边形P的边 e_i do
	{
	if 线 $x=x_0$ 和 e_i 产生相交 then 令此交点为(x_0, y_k);
	if $y_k < y_0$ then number=number +1; /*增加一个相交次数*/
	}
	Step 3: if number 是奇数 then 输出 "q 在多边形P内";
	else 输出 "q 在多边形 P 外";
	}

上述算法只需要 O(n) 时间执行（这里 n 代表多边形边的条数）。

9.3 天空轮廓

下一个例子是找出天空轮廓（skyline）问题，如表9.3所示。

表 9.3 天空轮廓问题

问题	输入一个城市中建筑物的位置和形状（建筑物都为矩形）。设计一个算法，找出这些建筑物在天空中所形成的轮廓。此轮廓需去掉所有隐藏线
输入	建筑物若干栋。所有建筑物都坐落于同一个水平线上。每栋建筑物由3个坐标(L, H, R)表示，其中L和R分别是建筑物的左右x轴坐标，而H是建筑物的高度 (1, 5, 8), (5, 8, 10), (7, 3, 11), (12, 2, 24), (17, 11, 19), (18, 4, 22)
输出	天空中的轮廓。此天空中的轮廓是一个人站在这些建筑物旁向远方看去，所见到的外围形状，由一连串从左到右的x轴坐标和高度交替构成(以下粗体的数字代表高度) (1, **5**, 5, **8**, 10, **3**, 11, **0**, 12, **2**, 17, **11**, 19, **4**, 22, **2**, 24, **0**)

天空轮廓问题在表9.3中的输入和输出可表示成图9.5和图9.6。注意在图9.6中，落在其他建筑物（矩形）内的线段都被删除了，只保留了最外围的线段。

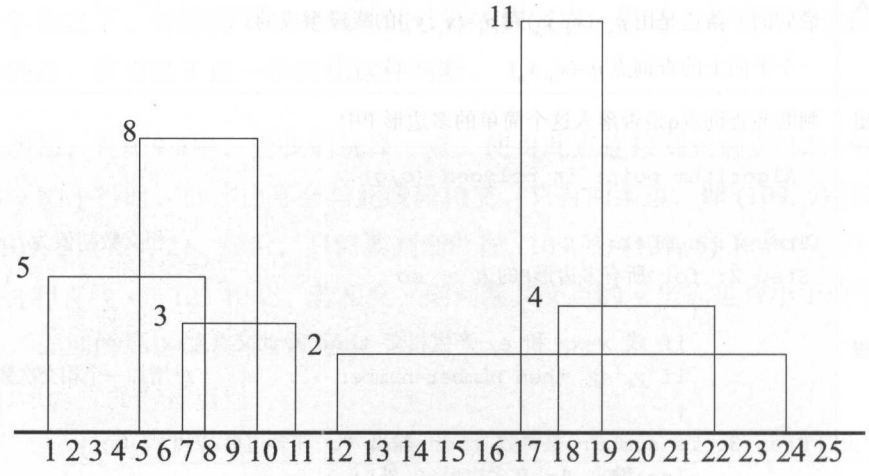


图9.5　表9.3中的天空轮廓问题的输入范例

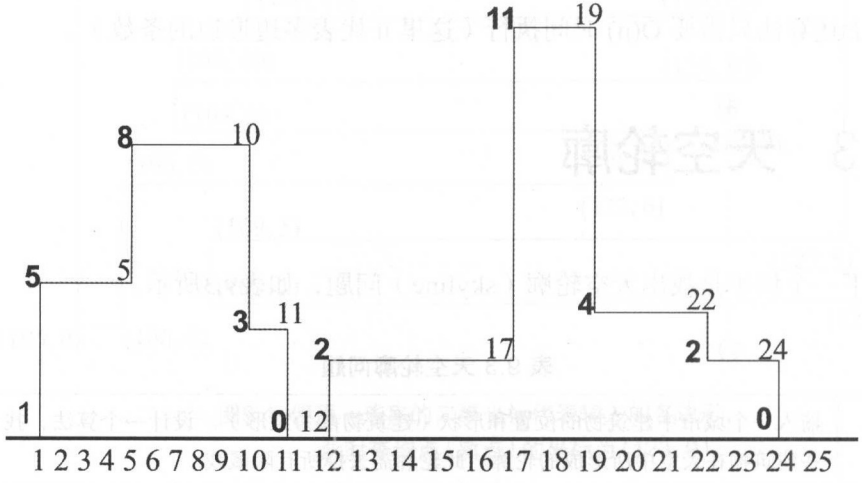


图9.6　表9.3中的天空轮廓问题的输出范例（粗黑数字代表高度，非粗黑数字代表x轴坐标）

解决天空轮廓问题最简单的方法是将建筑物一栋一栋地加入，并且在每加入一栋建筑物后，立刻调整其天空轮廓。如此在所有建筑物处理完后，即可得到最后的天空轮廓。为了完成此算法，假设在原来的天空轮廓上加入一栋建筑物，接下来观察如何调整其天空轮廓。

例如，在图9.6上加一栋建筑物 (7, 6, 21)，如图 9.7 所示。首先观察原来的天空轮廓 (1, 5, 5, 8, 10, 3, 11, 0, 12, 2, 17, 11, 19, 4, 22, 2, 24, 0) 会发生怎样的变化。天空轮廓显然被破坏了，尤其是 x 轴坐标介于 7 到 21 之间（即新加入建筑物的宽度范围）的轮廓若低于 6（即新加入建筑物的高度），则需要被隐藏起来，并且其高度需要被修正成 6。另外，建筑物的左右两道墙也有可能改变天空轮廓。

在图9.7中，建筑物的左墙被原先的轮廓隐藏了，但建筑物的右墙形成了新的天空轮廓，而建筑物的中间部分被调整为至少大于等于 6 以上。

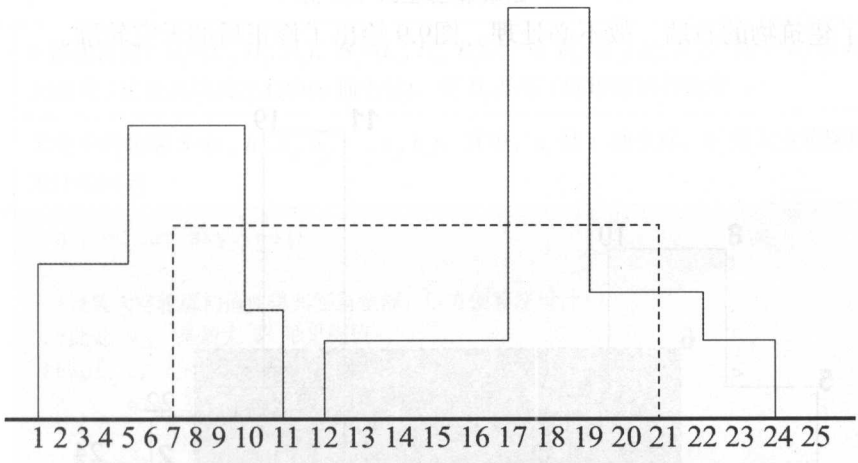


图9.7　在图9.5上加一栋建筑物 (7, 6, 21) 后，天空轮廓被破坏了

我们可以设计一个算法从左到右扫描整个天空轮廓并进行调整。首先，找到新加入的建筑物 (7, 6, 21) 的左墙位置 (即 x 轴坐标为 7)，接着逐一调整 x 轴坐标 7~21 之间的高度。

第一条碰到的平行线段为 5, 8, 10。因为此线段从 x 轴坐标 5 到 10，且其高度为 8，大于新加入建筑的高度 6，故不需调整。

第二条平行线段为 10, 3, 11，因为其高度低于新加入建筑的高度 6，故需改成 10, 6, 11。同样的理由下两条平行线段 11, 0, 12 需改成 11, 6, 12，而 12, 2, 17 需改成 12, 6, 17。这3条线段 10, 6, 11、11, 6, 12、12, 6, 17 因为高度相同，所以可合并成 10, 6, 17。

下一条平行线段 17, 11, 19的高度超过新加入建筑的高度 6，故不必调整。

最后一条平行线段 19, 4, 22 已经超过建筑物右墙（其 x 轴坐标为 21），因为其高度仍低于新加入建筑的高度 6，所以要将 19, 4, 22（可看成 19, 4, 21, 4, 22）改成 19, 6, 21, 4, 22。

天空轮廓的修正过程示意图如图9.8所示。最后剩下的天空轮廓因为早已经超过了建筑物的右墙，故不必处理。图9.9 绘出了修正后的天空轮廓。

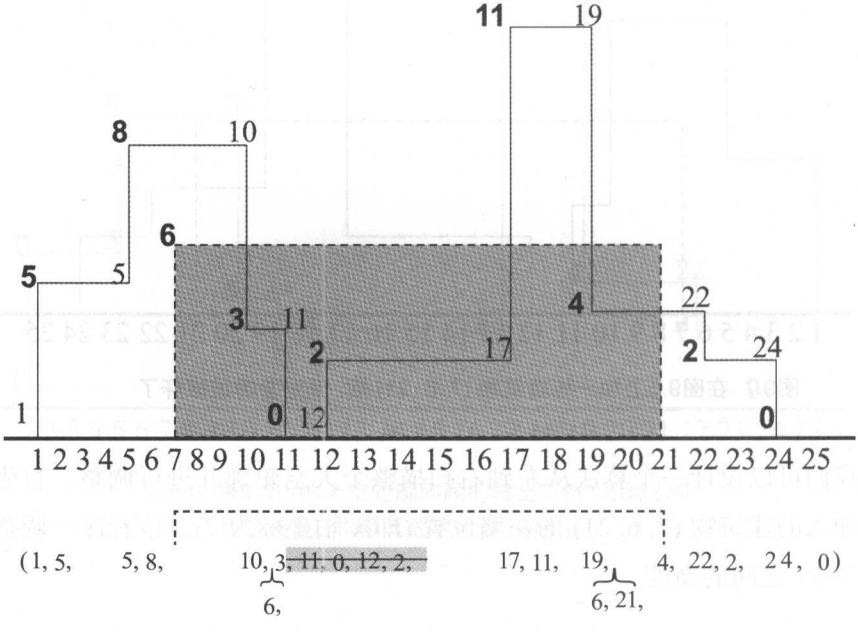


图9.8 天空轮廓的修正过程示意图

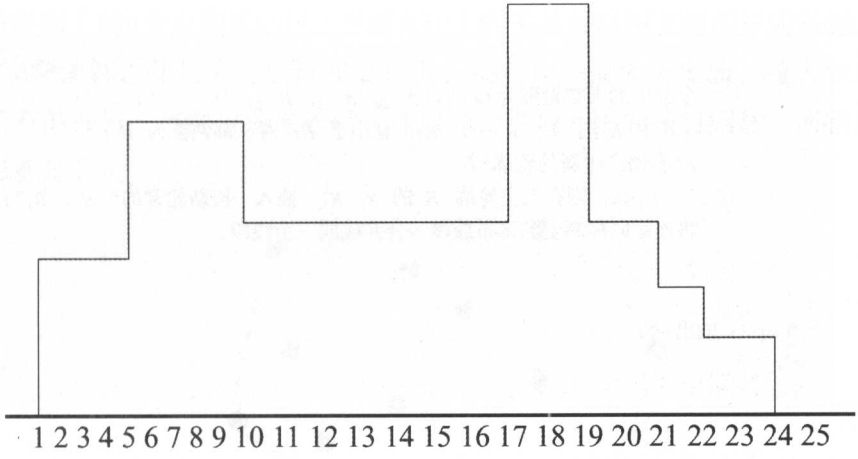


图9.9 修正后的天空轮廓 (1, 5, 5, 8, 10, 6, 17, 11, 19, 6, 21, 4, 22, 2, 24, 0)

详细的天空轮廓算法如表9.4所示。

表 9.4 天空轮廓算法

输入	n 栋建筑物：$B_1=(L_1, H_1, R_1)$，$B_2=(L_2, H_2, R_2)$，…，$B_n=(L_n, H_n, R_n)$，其中 L_i 和 R_i 分别是第 i 栋建筑物的左右墙(x 轴坐标)，而 H_i 是第 i 栋建筑物的高度
输出	天空中的轮廓 $S=(x_1, h_1, x_2, h_2, \cdots, x_z, h_z)$。其中，$x_i$ 是 x 轴坐标，h_i 是天空轮廓的高度($1\leqslant i\leqslant z$)
步骤	Algorithm skyline() { /*设置天空轮廓初值使得其涵盖全部，以方便算法设计*/ /*此处 x_{max} 是最大 x 轴坐标值*/ Step1: if L_1=1时，令$S=(L_1, H_1, R_1, 0, x_{max}, 0)$ else令$S=(1, 0, L_1, H_1, R_1, 0, x_{max}, 0)$; /*依序将 $B_i=(L_i, H_i, R_i)$加入到当前的天空轮廓 S 中*/ Step2: for i=2 到 n do { /*自左向右扫描 $S=(x_1, h_1, x_2, h_2, \cdots, x_z, h_z)$，并找到左墙的位置*/ 自左向右读取的天空轮廓S中一个片段 x_j, h_j, x_{j+1} 使得 $x_j\leqslant L_i<x_{j+1}$; /*左墙产生新的轮廓*/ if $h_j<H_i$ 则在天空轮廓 S 的 h_j 后插入一段新轮廓的"L_i, H_i"; /*修正中间的部分*/ 读取的天空轮廓S连续多个片段到 $x_k\leqslant R_i<x_{k+1}$ do

（续表）

步骤	{
	令其中的天空轮廓为(x_{j+1}, h_{j+1}, x_{j+2}, h_{j+2}, …, x_k)；
	若其中 h_{j+1}, h_{j+2}, …, h_{k-1} 有小于 H_i 者，都调整为 H_i；
	/*右墙产生新的轮廓*/
	if $h_k<H_i$ 则在天空轮廓 S 的 x_k 后，插入一段新轮廓的" H_i, R_i";
	将连续同样高度的多条线段，合并成同一个线段；
	}
	}
	Step3：输出 S;
	}

天空轮廓算法需要 $O(n^2)$的时间复杂度来扫描天空轮廓并进行适当的调整，此处的 n 是输入建筑物的个数。

9.4　凸包

下面介绍凸包（convex hulls）问题。凸包问题是一个十分有名的计算几何问题，如表9.5所示。

表 9.5 凸包

问题	一位制作皮件工厂的老板希望将每张皮革上的瑕疵点全数剪除，但是为了节省皮料，希望被剪除的整块面积（为凸多边形）越小越好。设计一个算法，协助老板进行此剪裁工作
输入	平面上的n个点 $P=\{p_1, p_2, \cdots, p_n\}$ P={(1, 5), (2, 12), (3, 8), (4, 4), (5, 6), (6, 11), (7, 1), (8, 10), (10, 7), (11, 13), (12, 3), (14, 9)}
输出	一个包含所有输入点的最小凸多边形（convex polygon），称作凸包C 。按照逆时钟方向，将凸包 C 上的点按序输出 C: (7, 1)→(12, 3)→(14, 9)→(11, 13)→(2, 12)→(1, 5)

平面上的n个点和其凸包（见图9.10）的关系可以用下面的比喻解释。就像给了n颗铁钉，并且将它们钉在黑板上，接着找一根大橡皮筋，拉大到足以包含所有的铁钉，将橡皮筋放松后，橡皮筋会被最外围的铁钉圈住，所得到的多边形就是凸包。

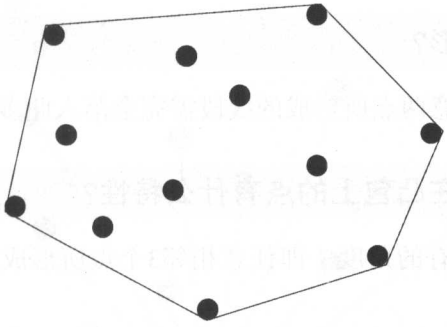


图9.10　平面上的点及其凸包

落在凸包上的点有什么特性?

好像这些点都位于最外围的地方。

为什么凸包上的点都在最外围呢?

凸包需要包含所有输入点，因此凸包被撑到最外围。

有些点也在非常靠外面的地方，却不在凸包上，如图9.11浅色圆点所示，为什么?

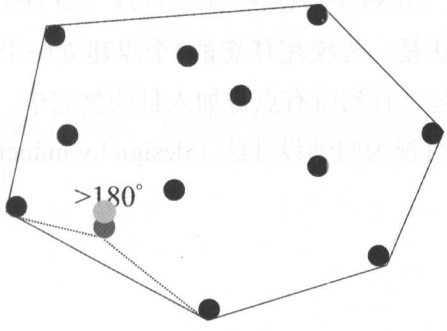


图9.11　浅色圆点（左下方）不在凸包上

因为会产生凹多边形的关系吧！

怎样判断出现了凹多边形？

看角度。从内部来看，有一个角度大于 180 就是凹多边形。

什么是凸多边形？

凸多边形内的任意两点所形成的线段需完全落入此多边形中。

从角度上看，在凸包上的点有什么特性？

从外面来看，所有的角度（即任意相邻3个点所形成的角度）都需小于 180 度。

最直接的凸包算法是：先使用任意的3个点建立一个凸包，每加入一个新点，就调整成新的凸包，直到所有点被加入且调整完毕。此类算法包括天空轮廓算法（见表9.4），可视为归纳设计法（design by induction），是一种常见的算法设计策略。

如果将上述方法稍微改变一下处理点的顺序，就会成为一个知名的凸包算法，称为格雷厄姆扫描（Graham's scan）。格雷厄姆扫描首先挑出最低的一点 p_1（若这样的点有多个，则选择其中最右边的点），计算此点和其他每一点连成直线的角度，并利用此角度将所有剩下的点排成逆时钟的顺序 $p_2, p_3, \cdots, p_n$（见图 9.12），以便后续处理。

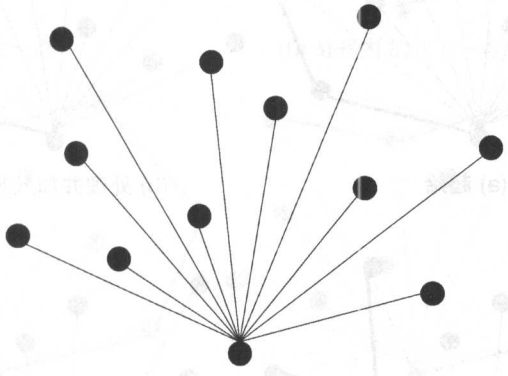


图9.12 平面上的每一点与最低点所形成的直线

假设处理前面 i-1 个点 $p_1, p_2, \cdots, p_{i-1}$ 后，所得到的凸包为 $c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{k-1} \rightarrow c_k$。

当格雷厄姆扫描处理下一个点 p_i 时，将会进行修正以找到新的凸包。首先，计算 pi 和最近凸包上的两点 c_k, c_{k-1} 所形成的角度，即 $\angle p_i c_k c_{k-1}$（此角度需从凸包内部测量）。如果 $\angle p_i c_k c_{k-1}$ 小于 180 度，就将 p_i 加入原来的凸包中，形成新凸包 $c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{k-1} \rightarrow c_k \rightarrow c_{k+1}=p_i$，并结束此点的处理。反之，如果 $\angle p_i c_k c_{k-1}$ 大于或等于 180 度，就将点 c_k 从凸包中移出，并继续对此凸包进行同样的检查，直到找到小于180度的角（并将 p_i 加入）为止。

格雷厄姆扫描重复以上步骤，直到所有点被处理完为止。图9.13所示为格雷厄姆扫描的一个范例。

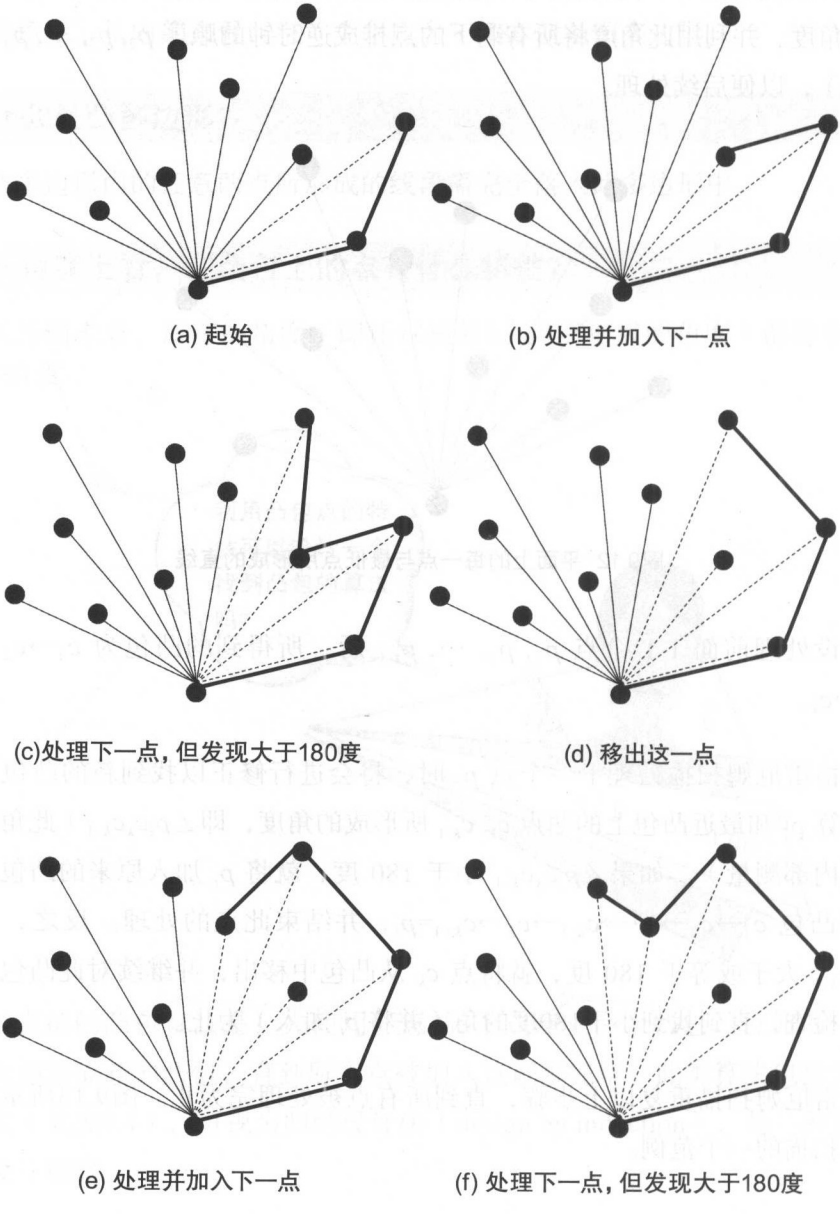
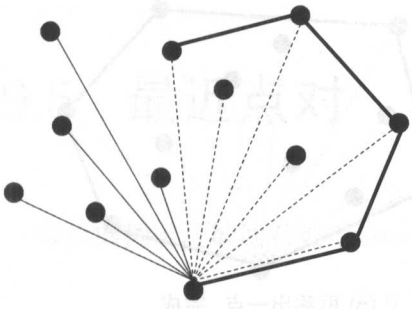
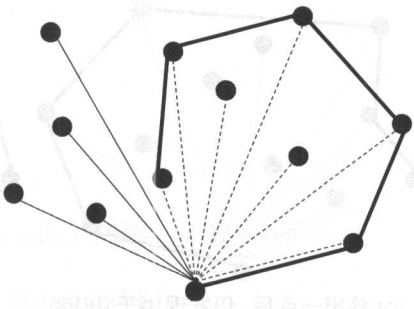


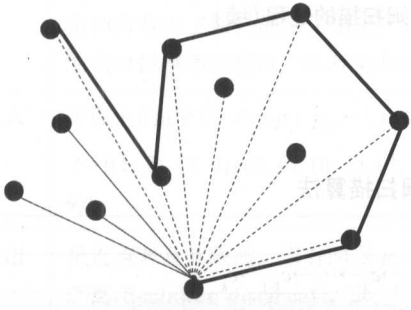
图 9.13 格雷厄姆扫描的过程



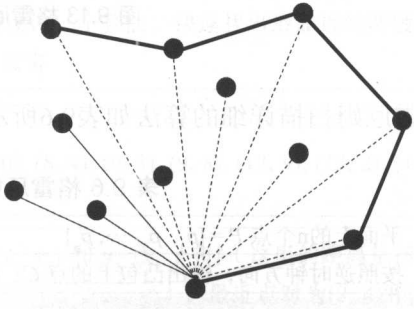
(g) 移除一点



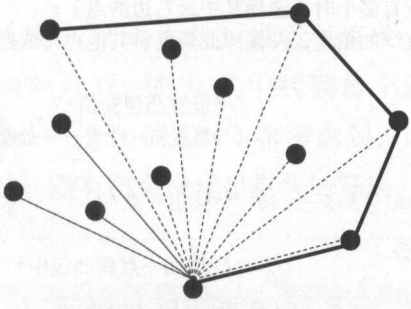
(h) 处理并加入下一点



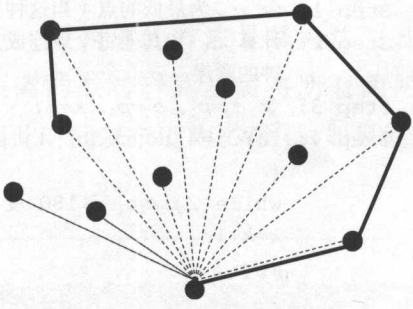
(i) 处理下一点，但发现大于180度



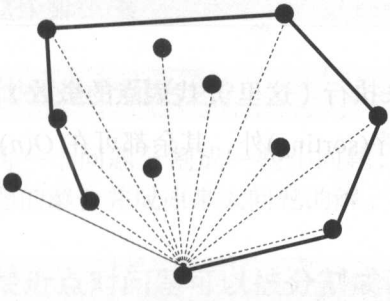
(j) 移出一点后，仍发现大于180度



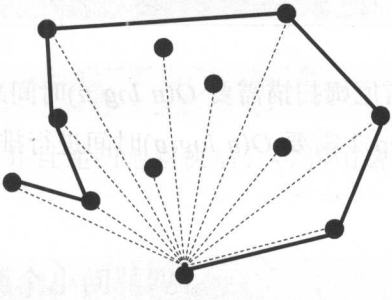
(k) 再移除一点



(l) 处理并加入下一点

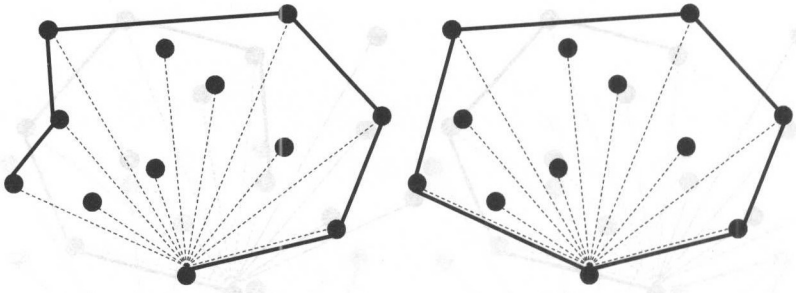


(m) 处理并加入下一点



(n) 处理下一点，但发现大于180度

图 9.13 格雷厄姆扫描的过程（续）



(o) 移出一点后，仍发现大于180度　　(p) 再移出一点，完成

图 9.13 格雷厄姆扫描的过程（续）

格雷厄姆扫描详细的算法如表9.6所示。

表 9.6 格雷厄姆扫描算法

输入	平面上的n个点 $P=\{p_1, p_2, \cdots, p_n\}$
输出	按照逆时钟方向，输出凸包上的点 C：$c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{k-1} \rightarrow c_k$

步骤	Algorithm Graham's_scan (P)
	{
	Step 1: 令 c_1 为最低的点（当这种点有多个时，选择其中最右边的点）；
	Step 2: 计算 c_1 和其他每个点连成直线的角度，并使用此角度将其他点排成逆时钟的顺序：p_2, …, p_n;
	Step 3: 令 $c_2=p_2$, $c_3=p_3$, $k=3$;　　/*设置凸包初值*/
	Step 4: for i=4 to n do　　/*将其余n-3个点一 一处理*/
	{
	while $\angle p_i c_k c_{k-1} \geqq 180$ 度 do {$k=k-1$;}; /*移出一点*/
	$k=k+1$;
	$c_k=p_i$;　　/*增加一点到凸包中*/
	}
	Step 5: 输出凸包 $c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_{k-1} \rightarrow c_k$;
	}

格雷厄姆扫描需要 $O(n \log n)$时间来执行（这里 n 代表点的个数），因为除了 Step 2 需要 $O(n \log n)$时间进行排序(sorting)外，其余都可在 $O(n)$时间内完成。

9.5　最近点对

最后的例子是最近点对（closest pair）问题，如表9.7所示。

表 9.7 最近点对问题

问题	在海上航行时，若两艘船不慎行驶太近，容易发生碰撞危险。假设我们可以及时收集到所有海上船只的定位信息。请设计一个算法，快速找出最靠近的两艘船，以方便实时提醒相关船员，以避免发生灾害
输入	平面上的n个点 $P=\{p_1, p_2, \cdots, p_n\}$ $P=\{(1, 7), (2, 4), (5, 6), (6, 1), (7, 10), (8, 4), (9, 3), (9, 8), (12, 9), (13, 2), (14, 5), (14, 9)\}$
输出	最近点对及其距离。当有两点 $p_1=(x_1, y_1)$ 和 $p_2=(x_2, y_2)$时（这里的距离是指欧几里德距离(Euclidean distance)，即 $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$），最近点对为(8, 4)和(9, 3)，距离为$\sqrt{2}$

从 n 个点中找到最近的两个点，最简单的方法是暴力法：比较所有的两两点对，并从其中找到最小距离的点对。此法需将所有两两点对(共 $\binom{n}{2}=\frac{n\times(n-1)}{2}$ 种点对）都考虑到，因此时间复杂度为 $O(n^2)$。如果使用分而治之法，就有机会设计出更快的算法。

如何设计出有效的分而治之算法?

这个嘛……

什么是分而治之?

将一个问题分割成一些小问题，并且递归地解决后，再利用这些小问题的解合并成原来大问题的解。

最近点对问题可以被分割成两个小问题吗?

应该可以。根据 x 轴坐标将所有点排序后，从中间平分即可。

如何利用两个小问题的解合并成原来大问题的解？

分别找到一个最近点对后，从两者中再选出更近者即可。

没有其他点对被遗漏了？

好像没有，除非落在中间，并且此点对的两点正好落在分割线的不同端，如图9.14所示。

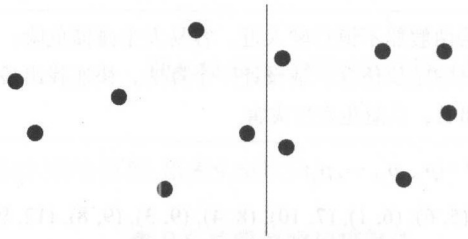


图9.14 利用分而治之法寻找最近点对时，需考虑两点落在分割线不同端的点对

这样的点对有多少对？

好像有很多可能。

每一种可能的点对都要考虑吗？

太远的点对应该不需要考虑。

为什么太远的点对不需要考虑？

因为要找的是最近的点对。

多远的点对不需要考虑？

只要距离超过两边选出的最小者，都可以不考虑，因为不会产生更近的点对。

跨越两边且距离小的点对个数多吗？会不会影响合并时的速度？

应该不会太多。因为这些点如果落在同一边，其间隔就需要大于一定距离，所以分布不会太密集，应该不会增加太多分而治之算法的运行时间。

设计一个分而治之最近点对的算法，关键在于如何合并和所需的运行时间。根据以上讨论，此合并需检查是不是有落在分割线两端且更靠近的点对。

当在左边找到的最小点对距离为 d1，而在右边找到的最小点对距离为 d2时，跨越两边的最小点对（如果存在的话）的距离不可超过 $d=\min\{d_1, d_2\}$。

若令此最小点对其中一点 p_L 位于左边，另一点 p_R 位于右边，则必可找到两个相邻且分处两边的 $d \times d$ 方格包含此最小点对。否则，p_L 和 p_R 的距离会超过 d，如图 9.15所示。

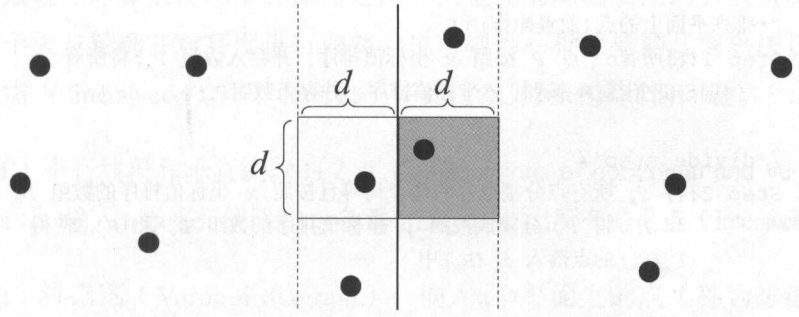


图9.15 跨越两边的最近点对的距离不可超过 $d=\min\{d_1, d_2\}$

落在同一方格中的点必须距离等于或超过 d ，否则在该边所找到的最近点对就是错的。所以，在左边同一方格中的点最多只有4个，且被“逼迫”退到4个角上。另外，在右边相邻的方格上也有同样的情况，即最多只有4个点，且被“逼迫”退到4个角上，如图 9.16所示。

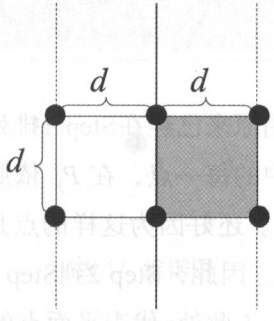


图9.16 跨越两边相邻的 $d \times d$ 方格内的最近点对最多只有6个点

这个分而治之的最近点对算法合并时在以分割线为中线的长为 2d 的带状范围内寻找可能存在的最近点对。寻找的方法是：在此范围中的每一点 p，检查是不是在一个方格的距离内有更靠近（距离比d小）的点对。如果有，就将此最近点对和距离返回。

详细的最近点对算法如表 9.8 所示。

表 9.8 最近点对算法

输入	平面上的n个点 $P=\{p_1, p_2, \cdots, p_n\}$
输出	最近点对及其距离

步骤	Algorithm closest_pair (P)
	{
	/*排列平面上的点(前置操作)*/
	Step 1:将所有n个点 P 按照 x 坐标值排序，并存入数组 P_x;将所有 n 个点 P 按照 y 坐标值排序，并存入数组P_y;
	
	/*divide step*/
	Step 2:将 P_x 所有点分割成左右两个均等且按照 x 坐标值排序的数组 L_x 和 R_x;将 P_y 分割成按照 y 坐标值排序的数组 L_y 和 R_y，使得 L_x(R_x)的点落入 L_y(R_y)中
	
	/*conquer step*/
	Step 3: 递归地计算左、右两集合的最近点对，令在左边找到的最小点对的距离为 d_1，令在右边找到的最小点对的距离为 d_2;
	
	/*merge step*/
	Step 4:令 d=min{d_1, d_2}，即找出 d_1 和 d_2 的最小值
	Step 5:自 P_y 中移出长为 2d 的带状范围（以分割线为中线）外的点后存入 $P_{y'}$;
	Step 6:按照 y 轴坐标值顺序扫描 $P_{y'}$,中的每一点，并计算此点和随后 y 轴坐标值的差小于d点的距离，记录扫描过程中发现的最近点对
	Step 7:从Step 3和Step 6所找到的点对中选择最近点对（及其距离）返回;
	}

上述算法的Step 2可以利用原来已经在Step 1排列好的数组 P_x 和 P_y在 $O(n)$ 时间内完成。Step 6 需为 $P_{y'}$ 中的每一点，在 $P_{y'}$ 依照此点的 y 轴坐标值增减 d 的范围内寻找可能的最近点对。还好因为这样的点是有限的（见图9.16），所以Step 6可在 $O(n)$ 时间内完成。因此，Step 2到Step 7的时间复杂度可用此数学式子表示：$T(n)=2T(n/2)+O(n)$（此处n代表平面点的个数）。解开此数学式子

可知，$T(n)$需要 $O(n \log n)$ 的时间来执行。最后，整个算法需要$T(n)+ O(n \log n)$（即 Step 1 的排序时间）$=O(n \log n)$时间来完成。

9.6　计算几何的技巧

归纳法（induction）和分而治之法（divide and conquer）都是计算几何上常见的技巧。例如，本章中的天空轮廓算法和格雷厄姆扫描算法使用了归纳法，而最近点对算法使用了分而治之法。天空轮廓算法使用的方法是从左到右扫描整个天空轮廓并对高度进行调整，也可视为一种重要的几何算法技巧，称为线扫描（line sweep）。下面列出一些其他常见的计算几何问题。

（1）平行线段和垂直线段的交点（intersections of horizontal and vertical line segments）：输入 n 条平行线段和 m 条垂直线段，找出所有的交点（intersection）。

（2）维诺图（Voronoi diagram）：输入n个平面上的点（称为维诺点），将平面分割成几个区域，使得每一个区域包含所有靠近其中一个维诺点的平面点，如图 9.17所示。

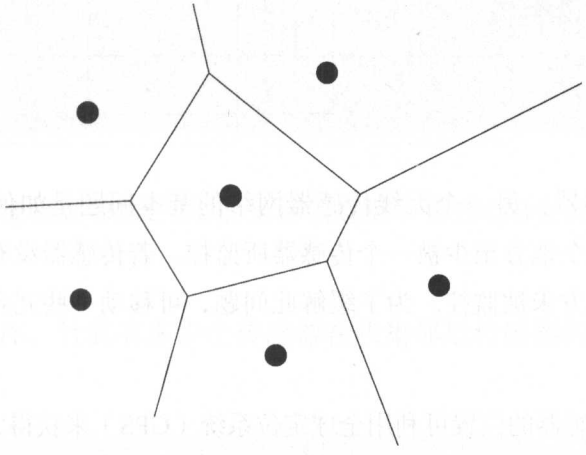
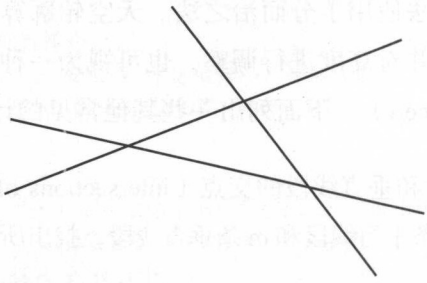


图9.17　维诺图

（3）最优多边形三角分割（optimal polygon triangulation）：将一个多边形切成多个三角形（切点须在顶点上且分割线不可相交），使其所需的分割线段长度总和最小。

学习效果评测

1. 平面上的线：一块比萨被连续直切数次之后，最多会得到几片？数学用语的描述是，一个平面可以被n条直线最多分割成几个区域？当 n=1 时，可分割成2个区域；当 n =2 时，最多分割成4个区域；当 n =3 时，最多分割成7个区域。

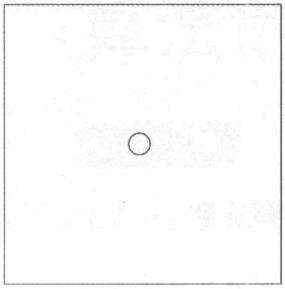


编写一个程序，当输入n时，计算最多的分割区域，并将此数输出。

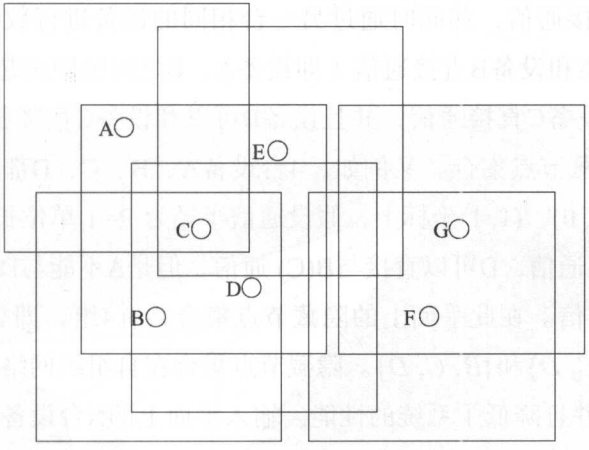
输入	
2	
3	
输出	
4	
7	

2. 决定冗余传感器：另一个无线传感器网络的基本问题是如何部署传感器网络，使得每一个地方至少被一个传感器所监控。若传感器没有均匀分布，则会导致若干地方未被监控。为了缓解此问题，可移动一些冗余传感器到未被监控的地方。

   假设每一个传感器的位置可利用全球定位系统（GPS）来获得，每一个传感器的通信半径（等同于其传感半径）都一样大。为了简化计算，我们再假设每一个传感器的通信（传感）范围是一个正方形，而其中心代表此传感器的位置。



每一个传感器随时记录它邻近传感器的坐标。例如，传感器C可以知道传感器 A、B、D、E，因为C落在 A、B、D、E 的正方形中。反之，传感器F不是C的邻居，因为C没有落在F的正方形中，所以收不到F的位置信息。在收到邻近传感器的坐标后，传感器C发现它自己是冗余的，因为它的监控区域可以被邻近传感器 A、B、D、E 完全覆盖。



相反地，当接收到邻居C和D的坐标信息后，传感器B会发现它自己不是冗余的。

编写一个程序，计算有多少个传感器在收集邻近传感器的坐标后会察觉自己是冗余的。

输入	
4	(传感器的个数)
6	(传感器的通信和监控半径)

```
4 4                    (以下为传感器的坐标)
6 4
8 4
10 4

输出
2                      (冗余传感器的个数)
```

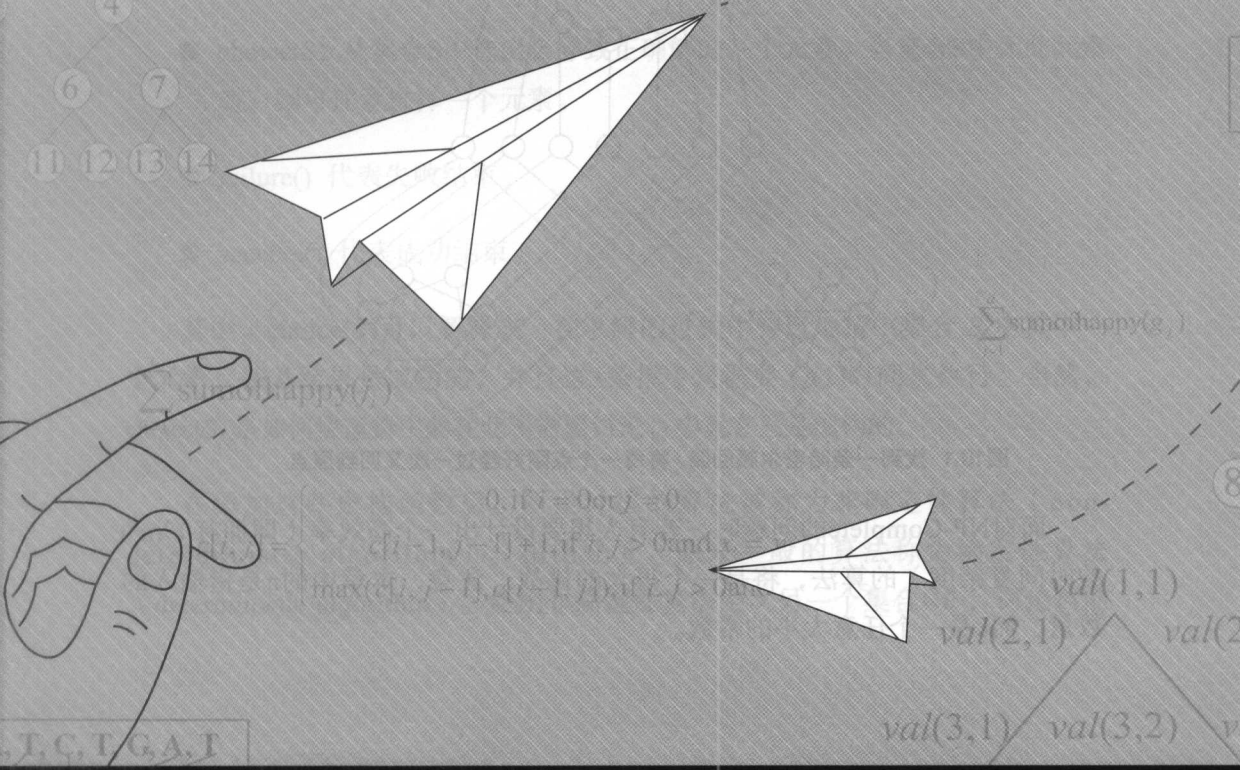
3. 隐藏节点问题：自组织网络（ad hoc networks）是一种随意连接，不需要基础网络建设的无线网络。每一台设备（device）在此网络上的功能如同一台路由器（router），可以寻找和维护网络路径。假设每一台设备的通信半径为 R 单位长度。当设备 A 和 B 的距离小于或等于 R (即$(x_1\text{-}x_2)^2+(y_1\text{-}y_2)^2 \leqslant R^2$，此处 (x_1, y_1) 和(x_2, y_2) 是 A 和 B 的坐标)时，设备A 可以直接和设备 B 通信。

   在自组织网络中，隐藏节点问题（hidden-terminal problem）的出现是因为两台设备无法直接通信，却同时通过另一台相同的设备进行数据的传送。例如，设备A不能和设备B直接通信（即设备A、B之间的距离超过R），但是设备A可以和设备C直接通信，并且设备B可以和设备C直接通信，这3台设备就形成了隐藏节点集合。又例如，4台设备A、B、C、D部署在平面的(0, 0)，(0, 1)，(1, 0)，(1, 1)坐标上，假设通信半径为 R=1 单位长度，如此A可以直接与 B(C) 通信，D可以直接与B(C) 通信，但是A不能与D直接通信，B不能与C直接通信。在此平面上的隐藏节点集合共有4组，即{A, B, C}、{A, B, D}、{A, C, D}和{B, C, D}。隐藏节点集合在自组织网络中严重地延长了通信时间，并且降低了系统的性能。输入平面上的N台设备，计算出有多少个不同的隐藏节点集合。

```
输入
4                      (部署设备的台数)
1                      (部署设备的通信半径 R)
0 0                    (以下是部署设备的平面坐标)
0 1
1 0
1 1

输出
4                      (不同的隐藏节点的集合数目)
```

算法的难题

章节大纲



10.1 什么是 NP-Complete

什么是 NP-Complete?

针对 NP-Complete 的问题，目前并没有 $O(n^k)$时间复杂度的算法被设计出来。也并没有被证明，这样的算法是不存在的。

NP-Complete（NP-完全或NP-完备）是一个集合，包含许多困难的算法问题。若将解决算法问题比喻成电脑游戏中的怪兽，这一类怪兽的战斗力是十分强大的，目前人类尚不能完全战胜它们。更不幸的是，NP-Complete 问题几乎遍及信息应用的所有领域。例如，哈密尔顿回路（Hamiltonian Circuit）问题（可否找到一条路径将每一个点刚好经过一次又回到原点）就是这样的难题，如图10.1所示。注：NP是非确定多项式的缩写，NP问题是指还未被证明是否存在多项式算法能够解决的问题。

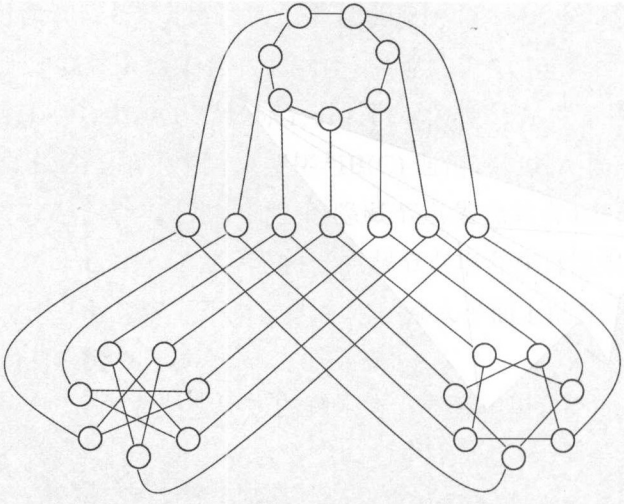


图10.1 找到一条哈密尔顿回路，将每一个点刚好经过一次又回到原点

面对NP-Complete的问题时，若有人想要设计出一个高效率（即拥有 $O(n^k)$的时间复杂度）的算法，将是一个很大的挑战。注意，本章中的n是指输入的数量，而k是一个任意大小的常数。

10.2 集合 P 和集合 NP

决策问题（decision problem）是指输出只有“是”或“否”的这类问题。例如，一个搜索问题：询问 x 是否出现在一个集合A中？若有，则输出“是”，否则输出“否”。因此搜索问题是一个决策问题。为了方便讨论，本章仅对决策问题进行难易分类。在决策问题中，一些较简单的问题有快的算法可以解决，但是也有一些较难的问题目前只存在慢的算法。

当一个决策问题存在 $O(n^k)$时间复杂度的算法时，称此问题落在P的集合中。落在P中的决策问题在本章中可以视为较简单的问题。

相对地，有一些决策问题目前尚无法将它们归入集合P中。为了思考这些问题，在一般算法可采用的功能上扩展了虚构的新指令。这些新指令虽然不存在于现实中，但是对探讨这些难题的性质和彼此的关系有很大帮助。以下是这些虚构的新指令：

- choice(S) 从集合S中选出会导致正确解的一个元素。当集合S中无此元素时，则可任意选择一个元素。
- failure() 代表失败结束。
- success() 代表成功结束。

其中，choice(S)可以解释成，在求解的过程中神奇地猜中集合 S 中其中一个元素，使其结果是成功的，并且这3条指令只需要 $O(1)$时间来执行。当然，choice(S) 是如何快速猜中的在此不需要讨论，毕竟它只是虚构的。

在添加这些虚构函数后，所设计出的算法被称为非确定性算法（nondeterministic algorithm）。相比之下，原来一般的算法称为确定性算法（deterministic algorithm）。使用非确定性算法定义另一个集合NP，讨论目前

尚无法归入集合 P 的难题。当一个决策问题存在一个 $O(n^k)$时间复杂度的非确定性算法时，称此问题落在NP的集合中。集合P和集合NP在本章中将扮演重要的角色。

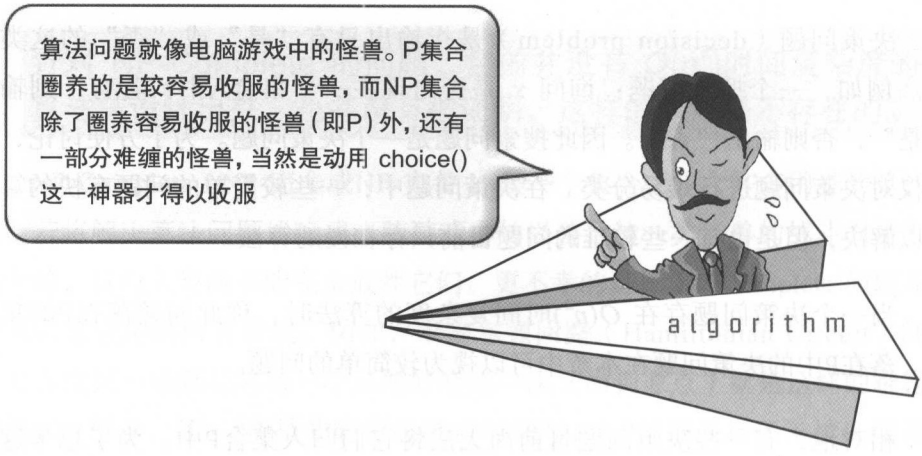


表10.1设计了一个非确定性算法来解决搜索问题。

表 10.1　搜索问题的非确定性算法

输入	存入 $A[1:n]$的 n 个值，其中 $n \geq 1$
输出	x 值在 $A[1:n]$中吗?

步骤	Algorithm search (A, x)
	{
	Step 1: j:=choice (1, n);　　/*利用 choice 直接猜中 x 的位置 j*/
	Step 2: if A(j)=x then {write (j); success ()};/*检查x 是否在 A(j)上*/
	Step 3: write (0); failure ();/*因 x 不在 A()上，输出搜索失败信息*/
	}

上述算法因为Step 1中使用了choice()，故为非确定性算法。其时间复杂度显然为 $O(1)=O(n^0)$，因此搜索问题落入 *NP* 中。相对地，设计出一个确定性算法，通过扫描数组来解此搜索问题时，可能需要 $O(n)$的时间复杂度，因此搜索问题也落入P中。

和非确定性算法相比，确定性算法常需要较多的时间来解决同一个问题。NP中的问题不会比P少（主要是在 $O(n^k)$时间内NP可以选择使用扩展的功能

choice() 的缘故）。简而言之，NP包含P，如图 10.2所示。

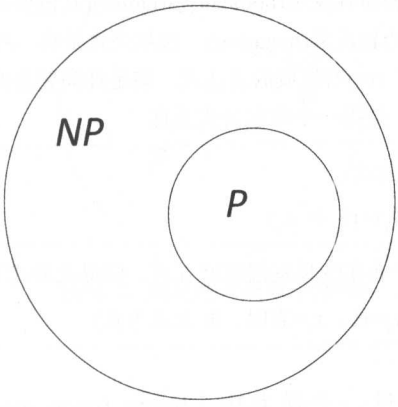


图 10.2　NP 包含 P

NP 内的问题都很难吗？

不！NP中也有简单的问题，例如搜索问题。

NP 比 P 大吗？

NP 看起来比 P 大，NP 包含 P。

有没有可能 NP = P？

不知道！可否告诉我答案？

目前还没有人可以证明 NP = P 或 NP ≠ P。

10.3　满足性问题

接下来介绍的满足性问题（Satisfiability Problem，SAT）就是一个NP中的典型难题，如表10.2所示。

表 10.2 满足性问题

问题	令 x_1，x_2，…，x_n 代表布林变量(boolean variables)(其值要么真(true)要么假(false)的变量)。令-x_i 代表 x_i 的相反数(negation)。布尔公式是将一些布尔变量及其相反数利用“与”(and)和“或”(or)所组成的表达式。满足性问题是判断是否存在指定每个布尔变量真假值的方式，使得一个布尔公式为真
输入	一个有n个变量的布尔公式 $(-x_1 \vee -x_2 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_2 \vee -x_1)$
输出	是否存在一种指定每个布尔变量真假值的方式，使得此公式为真? 是(当 x_1=真，x_2=真，x_3=真，x_4=真时，此公式为真)

我们可以很容易地设计一个暴力法（brute force method）解决满足性问题。此暴力法列出所有 n 个变量 2^n 不同的真假组合，并代入此式中，检测是否为真即可。可惜，此算法的时间复杂度为 $O(2^n)$，因此目前不能将满足性问题列入P集合中。相对地，表10.3将设计一个非确定性算法，并说明满足性问题可列入NP集合中。

表 10.3 满足性问题的非确定性算法

输入	有n个变量的布尔公式 $E(x_1, \cdots, x_n)$
输出	是否存在一种指定每个布尔变量真假值的方式，使得此公式为真

步骤	Algorithm satisfiability ($E(x_1, \cdots, x_n)$)
	{
	Step 1: for i=1 to n do
	　　x_i←choice (true, false) /*利用 choice 直接猜中 x_i 的真假值*/
	Step 2: if $E(x_1, \cdots, x_n)$ is true then success () /*计算此布尔公式是否为真*/
	　　else failure ();
	}

上述非确定性算法的时间复杂度为 $O(m+n)$。其中用 $O(n)$的时间来猜测 n 个变量的值，并用 $O(m)$的时间来计算长度为m 的布林公式 $E(x_1, \cdots, x_n)$是否为真。因此，满足性问题落入 NP 中。针对满足性问题，此非确定性算法比暴力法（确定性算法）减少了不少运行时间。综上所述，满足性问题这个难题目前不在P中，但是在NP中，如图 10.3所示。

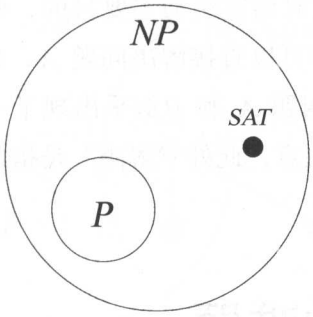


图10.3　满足性问题落于NP中

10.4　多项式时间转换

面对一个算法的题目时，可以利用问题转换（见第7章）将当前的问题转换成另一个问题。本节关心的问题转换技巧所需要转换的时间都需在多项式时间（即 $O(n^k)$）内完成。利用此多项式时间的转换可以将NP中的难题建立起一些有趣的关系。

针对两个问题 A 和 B，如果存在一个 $O(n^k)$时间的（确定性）算法，当且仅当问题B有解时，将每一个问题A的输入转换成问题B的输入，使得问题A有解。此关系被称为问题A转换成（reduce to）问题B，可表示成A∝B。

当 $A \propto B$ 时，若设计出一个 $O(n^j)$时间的确定性算法 β 来解决问题 B，则立即存在一个多项式时间的确定性算法可解决问题 A。原因是，我们可以先使用算法 α (需时间 $O(n^k)$)将问题 A 转换成问题 B，接着执行算法 β (需时间 $O(n^j)$)，因此共执行 $O(n^k)+ O(n^j)$的多项式时间，如图 10.4所示。

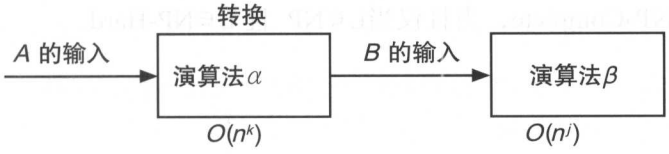


图10.4　多项式时间问题转换($A \propto B$)

若以设计出一个多项式时间算法为主要目的，则在多项式时间内将A 转换成问题 B有一个好处，即可以直接解决问题 A，也可以选择解决问题 B间接地解决问题 A。因此，问题 A 和 B似乎出现了一个依赖关系，即“解决B”→“解决A”的关系。注意，此处“解决”是指此问题存在一个 $O(n^k)$ 时间的确定性算法。

10.5 NP 中的难题

若将算法问题比喻成电脑游戏中的怪兽，NP-Complete 就是NP 中最难缠的怪兽。

首先定义一个名词，用来代表一组目前未被有效解决（指未落入P中）的算法问题。

当且仅当满足性问题转换成L（满足性问题 $\propto$ L）时，问题L被称为 NP-Hard（NP难解）。

我们已经知道了满足性问题是NP中的难题，而 NP-Hard 的问题是满足性问题衍生（转换）出来的。

换句话说，满足性问题这只怪兽进化成了另一只 NP-Hard 怪兽。

如果能有效地解决 NP-Hard 问题，就可以有效地解决满足性问题，即如果NP-Hard 问题落入 P 中，满足性问题也落入P中。

NP-Hard 问题虽然是从NP中转换而来的满足性问题，但不一定全部落于NP中，如图10.5所示。接下来讨论的是同时落在NP和NP-Hard中的问题。一个问题L被称为NP-Complete，当且仅当L∈NP 且 L∈NP-Hard。

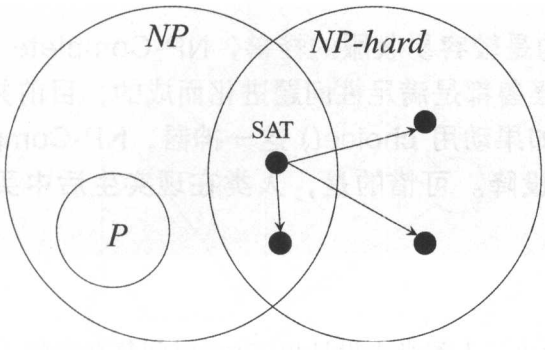


图10.5　NP-Hard问题是从满足性问题（SAT）转换过来的，但不一定落于 NP 中

NP-Complete问题从满足性问题转换而来，且落于NP中，如图10.6所示；可以想象成在NP中与满足性问题为同等级难度的怪兽。同样，如果能有效地解决 NP-Complete 的问题，就可以有效地解决满足性问题，即如果 NP-Complete 问题落入P中，满足性问题也落入P中。

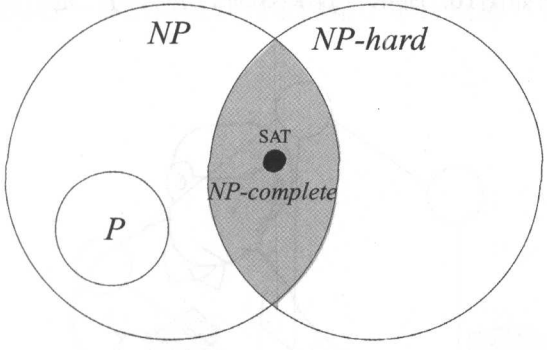


图10.6 NP-Complete(两集合交集处)是NP中的难题

简而言之，NP-Complete问题也是 NP-Hard 问题，两者都是从满足性问题转换而来的。差别是 NP-Complete 问题必须在 NP 中。

P集合圈养的是较容易收服的怪兽，NP-Complete 和 NP-Hard 两集合养的怪兽都是满足性问题进化而成的，目前并未被人类收服。但是，如果动用 choice() 这一神器，NP-Complete 内的怪兽就会立即投降。可惜的是，人类在现实生活中买不到这一神器。

截止到目前为止，人类尚未设计出 $O(n^k)$时间复杂度的（确定性）算法来解决 NP-Complete 或 NP-Hard 问题。更不幸的是，NP-Complete 或 NP-Hard 所涵盖的问题几乎遍布所有信息领域。下面介绍3个 NP-Complete 的范例。

1. 图着色（graph k-colorability）问题：此问题需回答“任意一个输入的图中，是否存在一种涂色方法，使用k种颜色使得相邻点上的颜色是不同的”。当输入的图如图10.7所示，且 k=2 时，答案为“否”。然而，当输入的图如图10.7所示，且 k=3 时，答案为“是”。

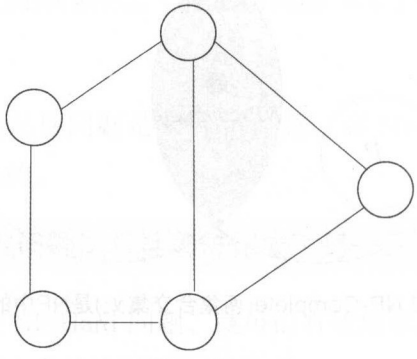


图10.7 图着色问题范例

2. 哈密尔顿回路（Hamiltonian circuit）问题：一个哈密尔顿回路为图上的一条路径，此路径需刚好经过每一个点一次，且形成回路（circuit）。哈密尔顿回路问题就是回答“任意一个输入的图中，是否存在一条这样的路径”。当输入的图如图10.8所示时，答案为“是”。

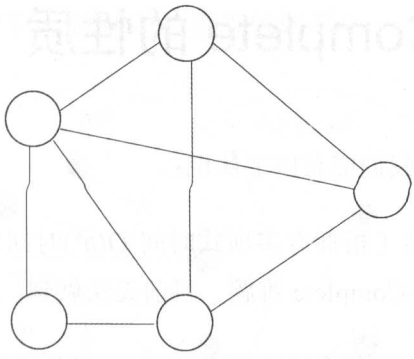
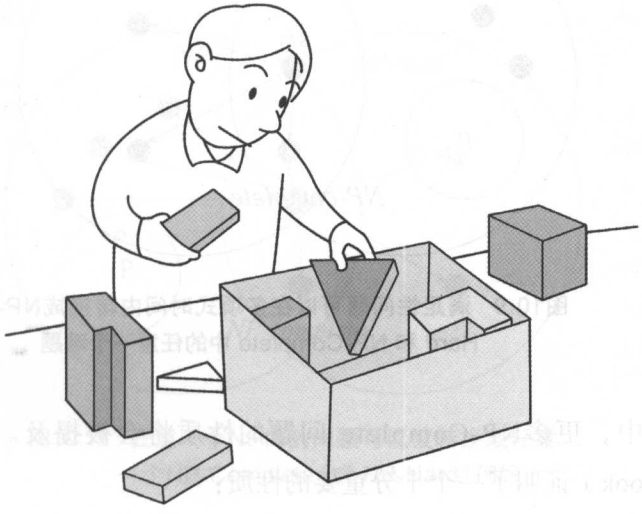


图10.8 哈密尔顿回路范例

3. 分割（partition）问题：输入一个正整数的集合。分割问题就是回答“是否可以将集合分割成两个小集合，使得每个小集合的总和是相同的”。当输入的集合为{2, 3, 4, 6, 9} 时，答案为“是”，因为 {2, 4, 6} 的和与 {3, 9} 的和是相同的。

另外，如何将一些东西装到固定的箱子中，也是一个 NP-Complete 的难题。



10.6 NP-Complete 的性质

根据之前的讨论，我们应有以下认识：

（1）想要有效率地（指拥有多项式时间 $O(n^k)$时间复杂度的确定性算法）解决一些 NP-Complete 难题，目前无法做到。

（2）利用 choice()，集合 NP 可以包含一部分这样的难题（如满足性问题）。

（3）多项式时间转换的关系可以被用来讨论这些难题之间的关系。

（4）满足性问题可以在多项式时间内转换成 NP-Hard 和 NP-Complete 中的任意一个难题，如图10.9所示。

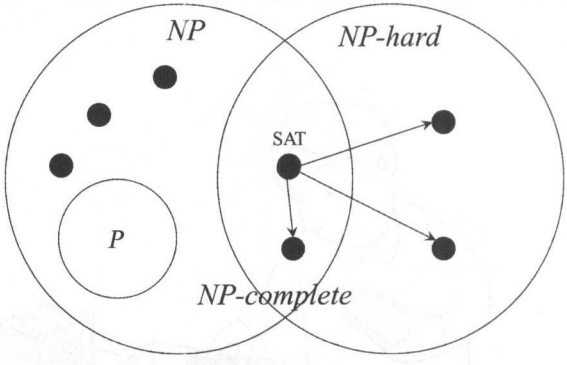


图10.9　满足性问题可以在多项式时间内转换成NP-Hard 和 NP-Complete 中的任意一个难题

在本节中，更多NP-Complete 问题的性质将会被提及。史蒂芬·库克（Stephen Cook）证明了一个十分重要的性质：

性质 (A)：任意一个 NP 内的问题都可以在多项式时间内被转换成满足性问题。

因此，满足性问题可以说是 NP 内最难的问题之一，如图10.10所示。

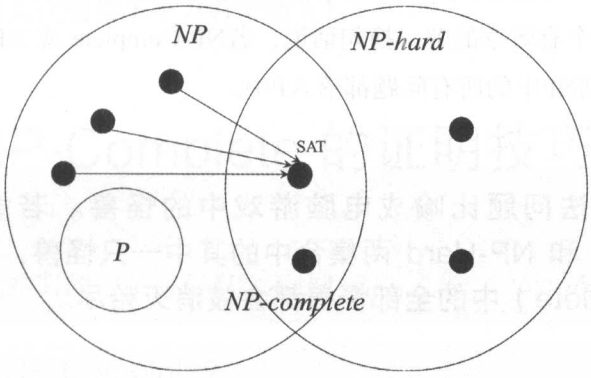


图10.10　任意一个 NP 内的问题都可以在多项式时间内被转换成满足性问题 SAT

因为满足性问题可以转换成 NP-Hard 和 NP-Complete 中的任意一个问题，使用性质(A)我们知道，任意一个NP内的问题可以在多项式时间内转换成任意一个NP-Hard 或 NP-Complete 中的问题，如图10.11所示。因此，我们可以得到性质(B) 和性质 (C)。

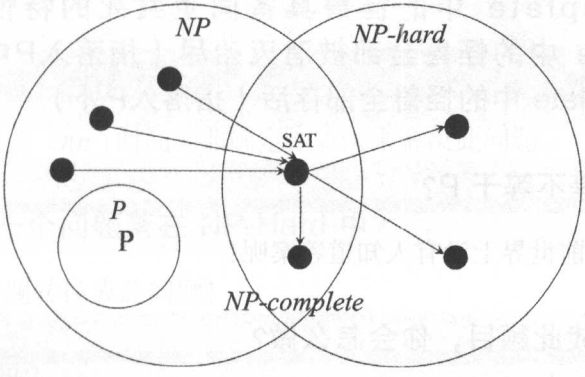


图10.11　任意一个NP内的问题可以被转换成任意一个NP-Complete（或 NP-Hard）问题

性质(B)：任意一个NP内的问题都可以在多项式时间内被转换成任意一个 NP-complete 问题。
性质(C)：任意一个NP内的问题都可以在多项式时间内被转换成任意一个NP-hard问题。

根据以上性质，若NP-Complete（或NP-Hard）其中一个问题被有效率地解决了（指设计出一个多项式时间 $O(n^k)$时间复杂度的确定性算法），则NP中的所有问题都将有一个有效率的解。换句话说，若NP-Complete 或 NP-Hard 其中一个问题落入P中，则NP中的所有问题都落入P中。

再次将算法问题比喻成电脑游戏中的怪兽。若能解决NP-Complete 和 NP-Hard 两集合中的其中一只怪兽，则 NP（含NP-Complete）中的全部怪兽都会被消灭殆尽。

因为满足性问题是 NP-Complete，故可推论性质(D)是正确的。

性质(D)：满足性问题在集合P中，当且仅当 P=NP。

相反地，若存在一个 NP-Complete 问题被证明不会落入P中，则所有 NP-Complete问题都不会落入P中（此时 P ≠NP）。

NP-Complete 中的怪兽具备同生共死的特性，即NP-Complete 中的怪兽全部被消灭殆尽（指落入P中），或者NP-Complete 中的怪兽全部存活（指落入P外）

到底 NP 等不等于 P?

不知道！目前世界上没有人知道答案呢！

如果要挑战此题目，你会怎么做?

为其中一个 NP-Complete 问题设计一个 $O(n^k)$ 时间复杂度的算法，或者证明其中一个 NP-Complete 问题绝对不存在此算法。

为何只需考虑一个问题就可以了?

因为 NP-Complete 问题彼此是好朋友，同生共死，绝不苟活。

问题好像变简单了，你的敌人只剩下一个了？

不过这个敌人代表着千千万万个敌人！

10.7 NP-Complete 的证明技巧

这个程序超难写的，怎么写都跑不快！

会不会是 NP-Complete？

不会吧？

怎么证明一个问题是 NP-complete？

不知道！

NP-Complete 的定义是什么？

落在 NP 且落在 NP-Hard 中。

怎么证明一个问题落在 NP 中？

设计一个花费 $O(n^k)$ 时间的非确定性算法来解决此问题。

怎么证明一个问题落在 NP-Hard 中？

将满足性问题转换成此问题！

转得过去吗？

这个……

其实，证明一个决策问题是 NP-hard不一定需要从满足性问题转换，比较可行的方式是从任意一个已经被证明为 NP-Hard 的问题下手。

理由是将一个 NP-Hard 的问题β转换成所关心的问题α时，我们可以推论问题α也是 *NP-Hard*。因为问题β 是 *NP-Hard*，故满足性问题必可转换成问题β，再加上之前发现的转换，满足性问题可以辗转转换成问题α，因此问题α也成为了 *NP-Hard*，如图 10.12 所示。

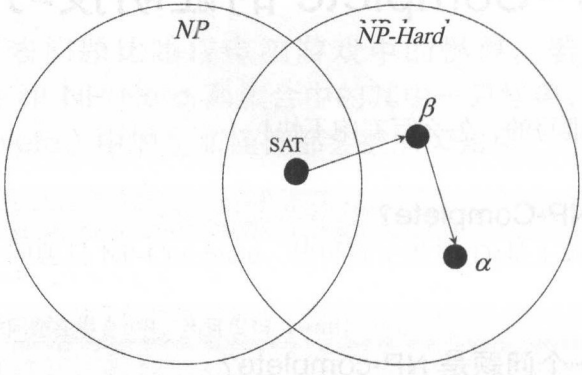
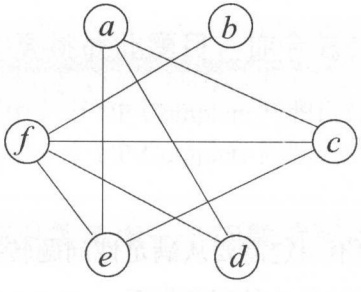


图10.12 从任意一个 NP-hard 问题(β)来证明另一个新问题(α)为 NP-Hard

下面我们将以顶点覆盖问题（vertex cover problem）说明此证明技巧，如表10.4所示。

表 10.4 顶点覆盖问题

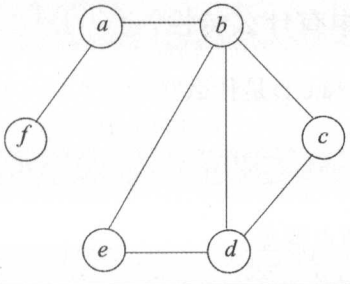
问题	网络攻击事件最近仍然频繁发生，给网络管理员带来极大困扰。网络管理员想在一些网络路由器(router)上部署追踪器(tracker)，从而记录追踪不正常的数据包，希望找到攻击者的网络所在。这些追踪器要求能够直接监控到每一条网络线
输入	一个图 G =(V, E) 代表网络路由器及其连接，k代表想要部署追踪器的个数 $G=(V=\{a, b, c, d, e, f\}, E=\{(a, c), (a, d), (a, e), (b, f), (c, e), (c, f), (d, f), (e, f)\})$，$k=3$ 

（续表）

输出	是否存在一个V的子集合 S，其 | S | =k，使得每一条E中的连线其中一个端点（endpoint）需落在 S 中？ 是(S={a, f, c})

已知 clique 问题是 NP-Hard问题，如表10.5所示。接下来准备将 clique 问题转换成顶点覆盖问题，借此证明顶点覆盖问题也是 NP-Hard问题。

表 10.5　clique 问题

问题	部分网络连接的线路断了，有时会造成通信中断。我们想要在一个网络中找出一个大小为k的子网，并期望此子网紧密连通不易断开。因此，希望其中任意两个节点有网络线直接连接。设计一个程序协助找出这样的子网
输入	一个图 G =(V, E)，k为正整数 G=(V={a, b, c, d, e, f}, E={(a, b), (a, f), (b, c), (b, d), (b, e), (c, d), (d, e)})，k=3 
输出	在V的子集合中是否存在一个大小为k的clique？此处V的子集合 S 被称为 clique。如果 S 中的任意两点在G上都有连线，那么(S={b, e, d})

此转换的重点在于，将 clique 问题的输入{G=(V, E), k}转换成顶点覆盖问题的输入{G'=(V', E'), k'}。转换的方法十分简单，新图的顶点集合和原来的相同，不同的是当原来的图上两顶点之间有线时，新图不需要连线；反之，当原来的图上两顶点之间没有连线时，新图需要连线，如图10.13所示。也就是两个图在顶点集合上是相同的，但在连线集合上是互补的，故称 G' 为 G 的互补图（complement graph）。最后，令 k'=|V|-k，转换完成。注意，此转换最多需要$O(n^2)$的时间。这里的 n 是顶点集合 V 的个数，因此是一个多项式时间的转换。

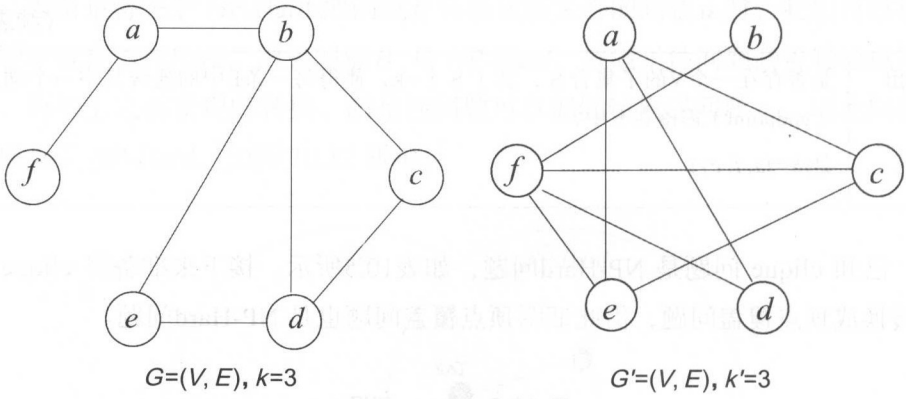


图10.13 将 clique 问题转换成顶点覆盖问题

在 G 中找得到大小为 3 的 clique 吗?

不难吧? {b, d, e}就是。

删除{b, d, e}后的顶点集合在 G'中有什么特性?

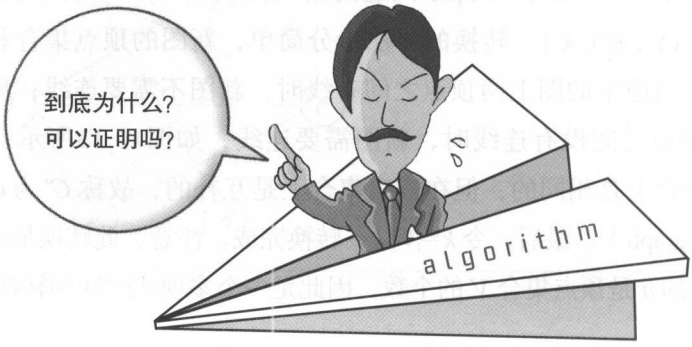
删除{b, d, e}后的顶点集合是{a,c,f}, {a,c,f}是什么?

在 G' 中{a, c, f}是顶点覆盖吗?

所有的线都粘到 a 或 c 或 f, 真的是顶点覆盖!

试试看其他 clique 是否正确。

删除 G 中的 clique {b, c, d} 后的顶点集合是 {a, e, f}, 所有 G' 的线都粘到 a 或 e 或 f, 也是顶点覆盖。



此转换的正确性可参考表10.6。最后，我们借助 clique 问题证明了顶点覆盖问题也是 NP-Hard。

表 10.6 clique 问题转换成顶点覆盖问题的正确性论述

当且仅当图 $G'=(V', E')$ 存在一个顶点覆盖，其大小为$\|V\|$-k时，图 $G=(V, E)$存一个 clique，大小为 k
证明： (1) 当图 $G=(V, E)$ 存在一个 clique $V^* \subseteq V$，大小为 k 时，V-V^*在补图 G' 中是一个顶点覆盖。 使用矛盾证明法，假设 V-V^* 在互补图 G' 中不是一个顶点覆盖，则在 G'上必有一条连线，其两端点 v_1、v_2 必落在 V^* 中。因为 G' 为 G 的互补图，故在图 G 中 v_1 和 v_2 必不相连，然而在图 G 中 v_1、v_2 都是 clique V^* 中的两个顶点，故在图 G 中，v_1 和 v_2 必相连，如此产生矛盾。因此，先前的假设错误。因此 V-V^* 在互补图 G' 中是一个顶点覆盖。 (2) 当图 $G'=(V', E')$ 存在一个顶点覆盖 $V^* \subseteq V'$，大小为 $\|V\|$-k时，V-V^* 在图 G 中是一个大小为 k 的 clique。 因为 V^* 在 G' 中是一个顶点覆盖，因此 E' 所有连线的两端点必定有一个落在 V^* 中。换句话说，V-V^*中的任意两个顶点必在 G 中是不连接的。又因为G' 为 G的互补图，故在图 G 中 V-V^* 中的任意两个顶点必是连接的。最后可知，V-V^* 是一个 clique，且大小为 $\|V-V^*\|=\|V\|-\|V^*\|=\|V\|-(\|V\|-k)=k$

学习效果评测

1. 排列：一个字符串abc的所有排列是 {"abc", "acb", "bac", "bca", "cab", "cba"}，按照字母顺序排列并给予编号：

```
0 "abc"
1 "acb"
2 "bac"
3 "bca"
4 "cab"
5 "cba"
```

输入

```
abc          (要按字母顺序排序后一个字符串)
3            (一个整数 n)
```

输出

```
bca          (编号为n的字符串)
```

2. 欧拉（Totient）函数：欧拉函数$\phi(m)$代表{1, …, m}中有多少个数和 m 互质？例如，$\phi(1)=1$, $\phi(2)=1$, $\phi(3)=2$, $\phi(4)=2$, $\phi(5)=4$, $\phi(6)=2$, $\phi(7)=6$。编写一个程序，当输入 m 时，输出其欧拉函数$\phi(m)$。

输入

```
13
8
```

输出

```
12
4
```

3. 是非题：填入答案 ○ 或 ×（若陈述不正确，说明原因）。

① (　　) NP的问题无法写出程序来解决。

② (　　) 所有P问题不一定可写程序找到解。

③ (　　) NP 不等于P。

④ (　　) NP=P。

⑤ (　　) NP-Complete 不是 NP 问题。

⑥ (　　) NP-Complete 不是 NP-Hard 问题。

⑦ (　　) NP-Hard 问题是 NP 中特别难的问题。

⑧ (　　) NP-Hard 问题都可以转换成 SAT 问题。

⑨ (　　) 无法编写程序有效解决的问题就是NP-Hard 或 NP-Complete。

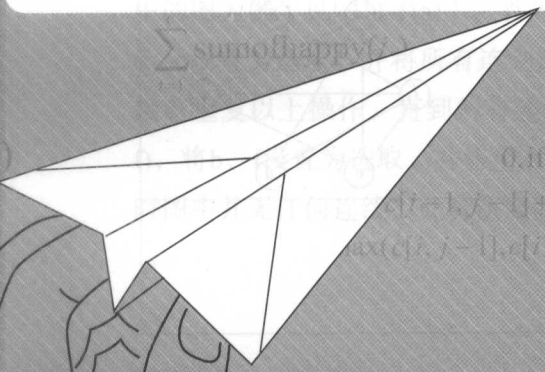
⑩ (　　) 若一个NP问题被有效解决了，则 NP=P。

逼近算法

章节大纲

$$\pi = \cfrac{4}{1+\cfrac{1^2}{3+\cfrac{2^2}{5+\cfrac{3^2}{7+\cfrac{4^2}{9+\ddots}}}}}$$

圆周率 π 的前一百个数字是 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164 06286 20899 86280 34825 34211 70679, 不过这只是逼近的数字而已。



11.1 什么是逼近算法

什么是逼近算法（approximation algorithm）？

简而言之，就是设计一个有效率的算法，找到与最优答案差距有限的解。

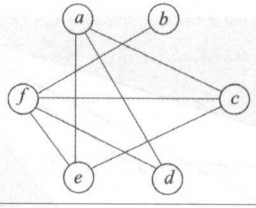
当碰到 NP-Complete 问题或其他难题时，想要找出最优解需要花费十分冗长的运行时间，有时花很长时间找到最优解反而不是一个适当的解法。退而求其次，倘若有一个算法可以很快地找到一个可以被接受的解，虽然不是最优解，但是与最优解差距不多，有时反而可以被接受。

一般而言，设计一个逼近算法需说明其解和最优解的差距有多大，以提供解答的质量保证。

11.2 最小顶点覆盖问题

第一个例子是最小顶点覆盖问题 (the minimum vertex cover problem)，如表11.1所示。

表 11.1 最小顶点覆盖问题

问题	给定一个图 $G=(V, E)$，找出图中最小顶点覆盖(vertex cover)。顶点覆盖为 V 的子集合 S，使得每一条 E 中的连线中的一个端点（endpoint）落在 S 中
输入	图 $G=(V, E)$ $G=(V=\{a, b, c, d, e, f\}, E=\{(a, c), (a, d), (a, e), (b, f), (c, e), (c, f), (d, f), (e, f)\})$ 
输出	V的子集合 S，符合以下两个条件： (1)每一条 E 中的连线中的一个端点落在 S 中 (2) $\|S\|$ (即 S 拥有的元素个数)需最小 $S=\{a, c, f\}$

最小顶点覆盖问题（见表11.1）和第10章提及的顶点覆盖问题（见表10.4）十分相似。差别在于最小顶点覆盖问题需找到最小的顶点覆盖集合并，输出此集合的顶点；而顶点覆盖问题只需判断是否存在一个大小为k的顶点覆盖（输出是或否）。目前已知顶点覆盖问题是 NP-Complete 问题，因此最小顶点覆盖问题也不容易有效地解决，本节将为此问题设计一个逼近算法。

什么是最小顶点覆盖问题？

找到最少的顶点，粘到所有的连线。

怎样的顶点可以粘到所有的线？

每条连线都需被所挑选的一个顶点粘到。

每一条连线有两个端点，应该挑哪一个呢？

不清楚！

每一条连线的两端点可以都不挑吗？

不可以。若是如此，则连接该两个顶点的连线将无任何顶点可照顾到。

需要两个顶点都挑选吗？

两个顶点都挑又太浪费了。

下面介绍的逼近算法希望找到足够少的顶点来粘住所有的连线。以表11.1中的图为例（见图11.1(a)），首先考虑图中任意一条连线 (a, c)。将其两个端点a、c设置为选取，并将所有连到这两个顶点的连线全部去掉（见图11.1(b)）。接着重复以上操作，直到所有连线都被去掉为止。下一条要考虑的连线为 (b, f)，将b、f设置为选取，并将所有连到b、f的连线全部去掉，得到图11.1(c)。此时图中并无任何连线，此算法结束。

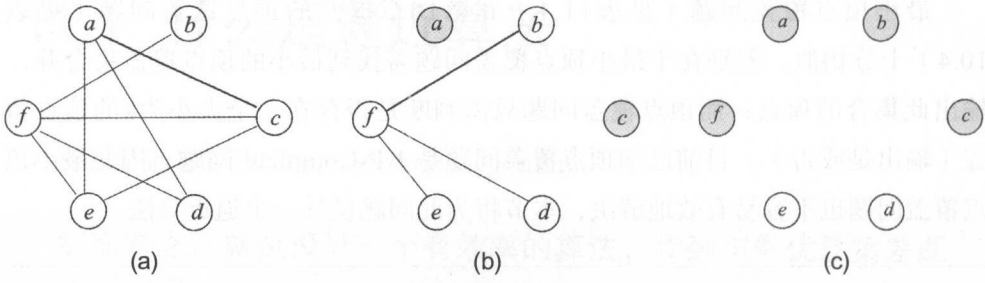


图11.1 最小点覆盖逼近算法的范例

在整个过程中，我们选取 {a, b, c, f}为顶点覆盖。虽然比最优解（如 {a, c, f}）所需的顶点多，但是此算法显然十分简单，且速度快。

此算法所得到的顶点覆盖数量不会超过最小点覆盖的两倍。原因是每一条被考虑的连线中，这个逼近算法会选取两个顶点，而最优解需在这两个顶点中至少选取一个顶点；否则这条被考虑的连线不会被最优解中任何顶点粘住（出现矛盾）。上述逼近算法如表11.2所示。

表 11.2 最小点覆盖逼近算法

输入	一个图 $G=(V,E)$
输出	顶点覆盖 S，使得 $\mid S \mid \leqslant 2\times S^*$，此处 S^*为最小点覆盖

步骤	Algorithm vertex_cover_approx ()
	{
	/*集合 S 存储选取的顶点，集合 C 记录的是剩下的连线*/
	Step 1: $S=\phi$;C=E;
	
	/*当集合 C 中有连线时，执行下列步骤*/
	Step 2: while $C\neq\phi$ do
	{
	自 C 中任意选出一连线(x, y);
	将顶点 x、y 加入 S 中;
	删除在 C 中和顶点 x、y 相连的连线;
	}
	Step 3: 输出 S;
	}

11.3 装箱问题

下一个例子是装箱问题（bin packing problem），如表11.3所示。

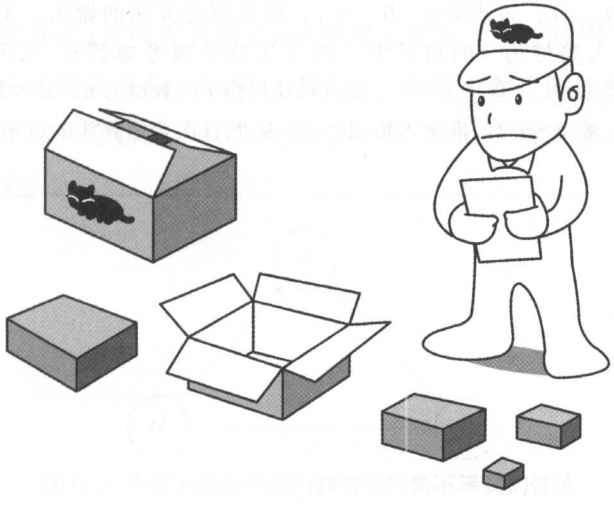


表 11.3 装箱问题

问题	老王任职于一家货运公司，担任包装工作。目前有一批不同重量的货物需要打包于同样规格的箱子中。此箱子的最大承受重量是固定单位，打包的过程只需要考虑重量因素。 请问老王应该如何将货物分开装箱才能使用最少的箱子，从而节省包装成本
输入	一批货物的重量为$\{w_1, w_2, \cdots, w_n\}$。每只箱子的最大承受重量为单位重量（即1），且每件货物的重量不超过单位重量(即$1<i\leqslant n$，$0<w_i\leqslant 1$) {0.2, 0.1, 0.5, 0.7, 0.4, 0.2, 0.3}
输出	最少使用箱子的数量为： 3 (即{0.2, 0.1, 0.5, 0.2}, {0.7, 0.3}, { 0.4})

装箱问题也是 NP-Hard，因此我们将专注于设计一个逼近算法。最直观简单的做法是利用首次合适（first fit）的概念：所有箱子按照从小到大的编号固定排列，每件货物依次按照此排列寻找第一个可以放入的箱子并直接放入。当输入为 {0.2, 0.1, 0.5, 0.7, 0.4, 0.2, 0.3} 时，需要3个箱子，且其装箱方式为 {0.2, 0.1, 0.5, 0.2}，{0.7, 0.3}，{0.4}。

下面将证明首次合适算法的解不会超出最优解的两倍。

假设执行首次合适算法后，装箱的情况为 $B_1, B_2, \cdots, B_m$(B_i 代表编号 i(此 $1\leqslant i\leqslant k$)，箱子的总重量 $B_i>0$)，共需要 m 个箱子。令任意相邻的两个箱子编号为 B_i 和 B_{i+1}，它们的和 $B_i+B_{i+1}>1$；否则若$B_i+B_{i+1}\leqslant 1$，则按照此算法的做法，编号 $i+1$ 箱子的货物会全部装入编号为 i的箱子中（因为比较早被考虑到）。又因为 $B_1+B_2>1$, $B_3+B_4>1$, …,$B_{\lfloor m/2\rfloor\times 2-1}+B_{\lfloor m/2\rfloor\times 2}>1$，最优解使用箱子的数量为 n，最少需 $\lfloor m/2\rfloor+1$(大于 $m/2$)个箱子，故 $n>m/2$。推导可得 $2n>m$，因此首次合适算法的解不会超出最优解的两倍。

11.4　平面上的旅行商问题

最后一个例子是平面上的旅行商问题（traveling-salesman problem），如表11.4所示。一般的旅行商问题是，在一个图中，找出一条最小成本的路径，使得此路径经过所有的顶点刚好一次，最后回到原起始点。本节讨论的平面上的旅行商问题稍有不同，需符合三角不等式（triangle inequality）的性质。

表 11.4　平面上的旅行商问题

问题	老王是一个勤快的推销员，每天需要拜访每一位客户刚好一次，并且最后需回到原出发的地点。为了节省时间，老王每次使用最短的直线距离前往下一个客户处。设计一个算法，帮他找到拜访所有客户一圈的最短旅程
输入	平面上n个顶点 $P=\{p_1, p_2, \cdots, p_n\}$ {(1, 2), (1, 4), (3, 3), (3, 5), (4, 1), (5, 4) }
输出	一个旅程 $p_{o(1)} \rightarrow p_{o(2)} \rightarrow \cdots \rightarrow p_{o(n)} \rightarrow p_{o(1)}$，整个旅程的距离总和最短。此处旅程中两点 $p_1=(x_1, y_1)$和 $p_2=(x_2, y_2)$ 的距离是指欧几里德距离(Euclidean Distance)，即 $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$ (1, 2)→(1, 4)→(3, 5)→(5, 4)→(3, 3)→(4, 1)→(1, 2)

平面上的旅行商问题中的顶点都在一个平面上，而且任意两顶点的距离（成本）为连接两者的直线距离。因为几何上的三角形有两边之长的和大于第三边的特性，故顶点a直接连到顶点b的距离（成本）小于或等于顶点a 绕过顶点c再连接顶点b的距离之和，如图11.2所示。所以平面上的旅行商问题符合三角不等式的性质。尽管如此，这个问题还是 NP-Hard，接下来将为这个问题设计一个逼近算法。

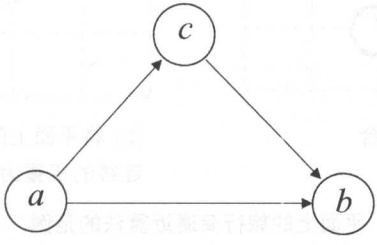


图11.2　平面上的旅行商问题符合三角不等式的性质

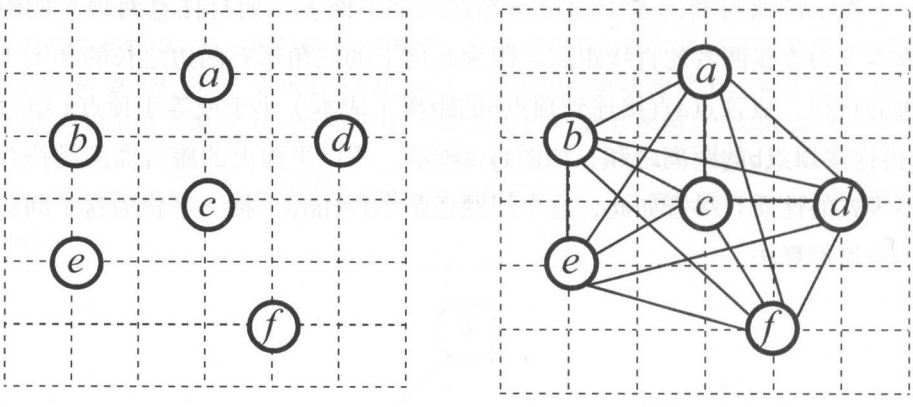
这个逼近算法先使用一棵树（tree）将所有顶点连接，再从此树上寻找一个拜访（或遍历）每一个节点（即顶点，但在树图中习惯称为节点）的旅程。此算法找到的解的距离总和不会超过最优解的两倍，如表11.5所示。

表 11.5　平面上的旅行商逼近算法

输入	平面上 n 个点 $P=\{p_1, p_2, \cdots, p_n\}$
输出	一个旅程 $p_{o(1)} \rightarrow p_{o(2)} \rightarrow \cdots \rightarrow p_{o(n)} \rightarrow p_{o(1)}$，整个旅程的距离总和不会超过最优解的两倍

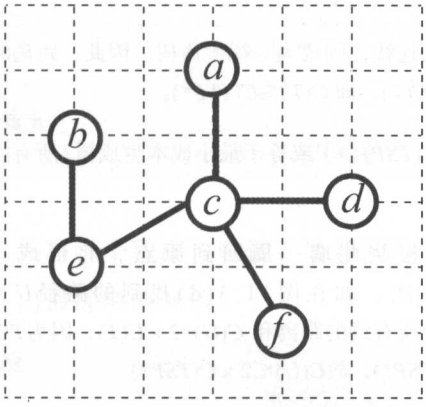
步骤	Algorithm TSP_approx ()
	{
	/*将平面上的顶点转成一个图*/
	Step 1: 建造一个图$G=(V,E)$，使得 V 中的每一个顶点 v_i 代表平面上的一个顶点 $p_i(1 \leqslant i \leqslant n)$，而且任意两个顶点$v_1$ 和 v_2 存在一条连线(v_1, v_2)落入 E 中，并令其权重为p_1 到 p_2的距离
	Step 2: 在图 G 上执行Kruskal 最小成本生成树算法，并产生最小成本生成树 T
	Step 3: 在 T上任意选一点v当作树根(root)，执行前序遍历法，并记录每个节点被拜访(遍历)的顺序$H=v_{o(1)} \rightarrow v_{o(2)} \rightarrow \cdots \rightarrow v_{o(n)}$，此处的 $v_{o(1)}$为 v
	Step 4: 输出旅程 $p_{o(1)} \rightarrow p_{o(2)} \rightarrow \cdots \rightarrow p_{o(n)} \rightarrow p_{o(1)}$
	}

上述算法中的前序遍历（preorder traversal）是指在一棵树中递归地遍历（即拜访）一个节点后，才开始遍历其儿子们。图11.3是上述算法的一个范例。

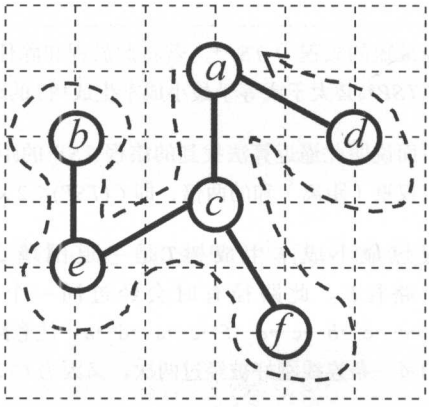


(a) 平面上的点集合　　(b) 将平面上的顶点转成一个图，且设置连线的权重为连接两个顶点的直线距离

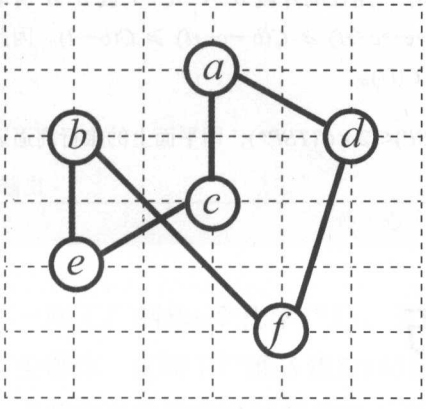
图 11.3　平面上的旅行商逼近算法的范例



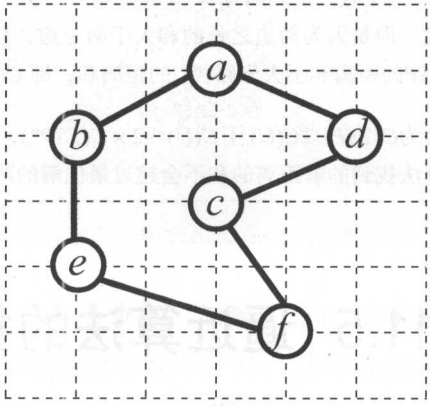
(c) 找出此图的最小成本生成树T



(d) 以a为起始点，列出T的前序遍历顺序



(e) 将T的前序遍历顺序最后连接回起始点a，形成一个完整的旅程



(f)此范例的最优旅程

图 11.3　平面上的旅行商逼近算法的范例（续）

最后证明平面上的旅行商逼近算法找到的旅程的距离和不会超过最优解的两倍。

令最短的旅程为 *TSP**。若将此旅程扣除任意一条连线，则成为一棵生成树。因此，距离和 $C(TSP^*)$必大于或等于最小成本生成树T的距离和 $C(T)$，即 $C(T) \leqslant C(TSP^*)$。

下面说明此逼近算法找到的旅程 *TSP* 的距离和 $C(TSP)$小于或等于最小成本生成树T所有连线权重（距离）和的两倍，即 $C(TSP) \leqslant 2 \times C(T)$。

假想最小成本生成树 T 是一道围墙，从树根绕此墙一周回到原点，将形成一个路径 E。此路径有时会经过同一个节点多次，如在图11.3(d)找到的路径 U 为 $a \rightarrow c \rightarrow e \rightarrow b \rightarrow e \rightarrow c \rightarrow f \rightarrow c \rightarrow a \rightarrow d \rightarrow a$。注意路径$U$中所有线的距离和 $C(U)=2 \times C(T)$，因为T中的每一条连线刚好被经过两次。又因为 $C(T) \leqslant C(TSP^*)$，故$C(U) \leqslant 2 \times C(TSP^*)$。

旅程*TSP*其实是在路径U中（除了头尾外）忽略重复经过的节点后所形成的旅程。例如，在图11.3(e)中，*TSP*是将路径 U：$a \rightarrow c \rightarrow e \rightarrow b \rightarrow e \rightarrow c \rightarrow f \rightarrow c \rightarrow a \rightarrow d \rightarrow a$忽略（跳过）重复节点e、$c$、$c$、$a$后，得到路径 $a \rightarrow c \rightarrow e \rightarrow b \rightarrow f \rightarrow d \rightarrow a$ 。其中，路径的片段 $b \rightarrow e \rightarrow c \rightarrow f$ 被 $b \rightarrow f$ 所取代。但是因为两边之长的和大于第三边，故 $C(b \rightarrow e \rightarrow c \rightarrow f) \geqslant C(b \rightarrow c \rightarrow f) \geqslant C(b \rightarrow f)$。因此*TSP*的距离和不大于路径$U$的距离和，即 $C(TSP) \leqslant C(U)$。

因为$C(TSP) \leqslant C(U)$且 $C(U) \leqslant 2 \times C(TSP^*)$，故$C(TSP) \leqslant 2 \times C(TSP^*)$，即平面上的旅行商逼近算法找到的解距离的和不会超过最优解的两倍。

11.5 逼近算法的技巧

一个逼近算法的解和最优解的差距需要被严格保证。若设计一个算法后，虽然执行后所得的解十分靠近最优解，但是不能提出严格的证明来保证每一次执行的质量，这种算法暂时不适合称为逼近演算法。当然，逼近算法的解和最优解的差距越小，就会更有价值。

学习效果评测

1. 传感器网络部署问题：输入一个 k × k的正方形，代表一个传感器网络部署的区域。请尽量部署最少数量的传感器（半径为r的圆）来覆盖（监控）整个区域。

输入

```
10                  (正方形的宽 k)
4                   (传感器的传感半径 r)
```

输出

```
4                   (部署传感器的个数)
```

2. 编写一个程序，验证在装箱问题中使用首次合适（first fit）算法的解不会超出最优解的两倍。

输入

```
7                   (货物的件数 n)
0.2                 (以下是这批货物的重量{w1, w2, …, wn}, 0<wi≤1)
0.1
0.5
0.7
0.4
0.2
0.3
```

输出

```
3                   (使用箱子的数量)
```

3. 一间工厂制作n个化学产品。有些化学产品不兼容，如果互相接触可能产生爆炸。这间工厂准备建造x间储藏室来存放这些化学产品，并且希望这些不兼容的化学产品被放置于不同的储藏室。为了决定需要兴建多少间储藏室，希望知道当n个化学产品被安全放置在x间储藏室中有几种不同的配置方式。

   注意每间储藏室都不同，而且可以存放任意多的兼容化学产品。如果有一些化学产品被安置在不同的储藏室中，两个配置方式将被视为不同。例如，针对3种化学产品 C_1、C_2、C_3 (n=3)和两间储藏室 P_1、P_2 (x=2)，其中 C_1、C_2不兼容，C_1(C_2)相容于 C_3。有以下4种配置方式：

   （1）$P_1=\{C_1\}$，$P_2=\{C_2, C_3\}$。

   （2）$P_1=\{C_2\}$，$P_2=\{C_1, C_3\}$。

   （3）$P_1=\{C_2, C_3\}$，$P_2=\{C_1\}$。

（4）$P_1=\{C_1, C_3\}$，$P_2=\{C_2\}$。

一般而言，针对 n 种兼容的化学产品和 x 间储藏室，有 x^n 种配置方式，x^n 被称为颜色多项式（chromatic polynomial）。相对地，针对 n种不兼容的化学产品和 x 间储藏室，可以有 $x(x\text{-}1)(x\text{-}2)\cdots(x\text{-}n\text{+}1)$种配置方式（此处 $x\geqslant n$）。同样地，$x(x\text{-}1)(x\text{-}2)\cdots(x\text{-}n\text{+}1)$为其颜色多项式。前面的范例所对应的颜色多项式为 $x^3\text{-}x^2$。因此，当储藏室的个数 x=2时，有 $2^3\text{-}2^2$=8-4=4种配置方式。

输入 n 种化学产品与其不兼容的关系。编写一个程序，输出其对应的颜色多项式。

输入

```
3              (化学产品的种类数)
1              (化学产品间不兼容关系的数量)
1 2            (以下是每一种不兼容关系)
```

输出

```
1 -1 0 0          (颜色多项式的系数)
```

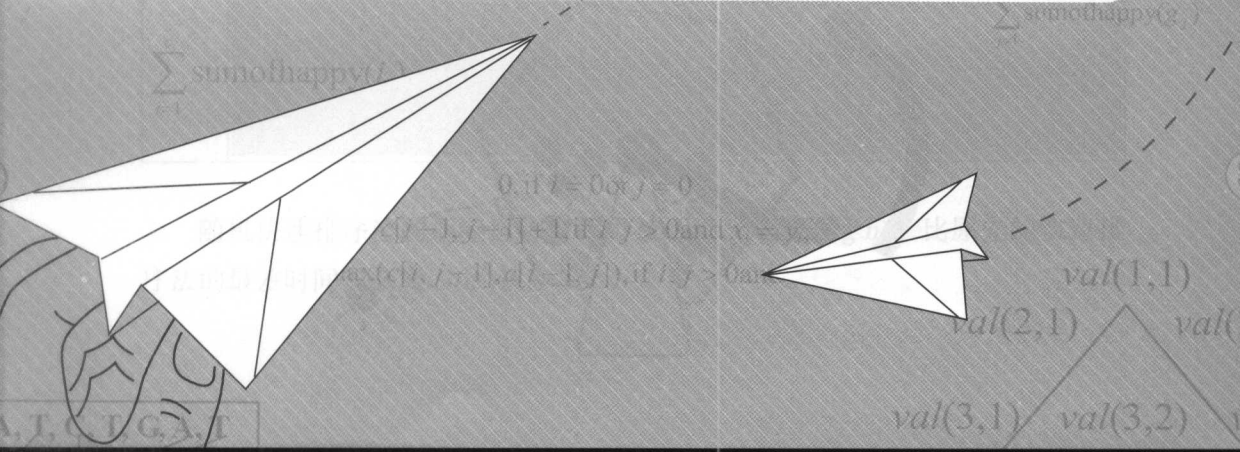
随机算法

章节大纲

虚竹慈悲之心大动，心知要解段延庆的魔障，须从棋局入手，只是棋艺低浅，要说解开这局复杂无比的棋中难题，当真是想也不敢想。眼见段延庆双目呆呆地凝视棋局，危机生于顷刻，突然间灵机一动："我解不开棋局，但捣乱一番，却是容易，只需他心神一分，便有救了。既无棋局，何来胜败？"，便道："我来解这棋局。"快步走上前去，从棋盒中取过一枚白子，闭了眼睛，随手放在棋局之上。

他双眼还没睁开，只听得苏星河怒声斥道："胡闹，胡闹，你自填一气，自己杀死一块白棋，哪有这等下棋的法子？"，虚竹睁眼一看，不禁满脸通红。原来自己闭着眼睛瞎放一子，竟放在一块已被黑棋围得密不通风的白棋之中。这大块白棋本来尚有一气，虽然黑棋随时可将之吃净，但只要对方一时无暇去吃，总还有一线生机，苦苦挣扎，全凭于此。现下他自己将自己的白棋吃了，棋道之中，从无这等自杀的行径。这白棋一死，白方眼看是全军覆没了。鸠摩智、慕容复、段誉等人见了，都不禁哈哈大笑。

金庸 天龙八部



12.1 什么是随机算法

什么是随机算法（randomized algorithms）？

简而言之，就是一个算法中含有随机决定的步骤。

天龙八部电视剧中的珍珑棋局如图12.1所示。

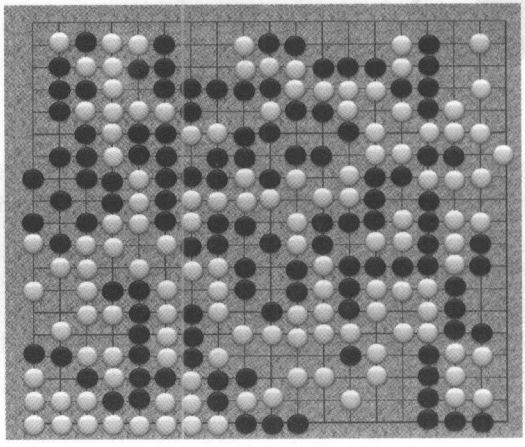
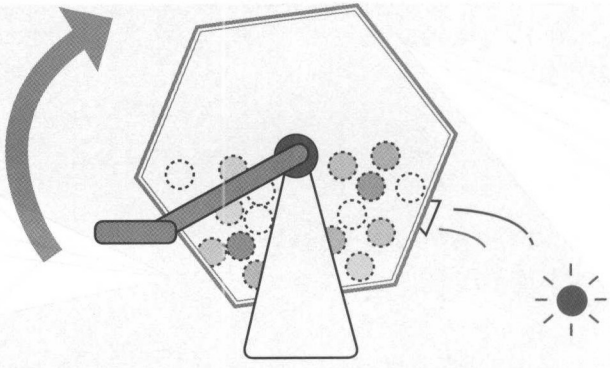


图12.1 天龙八部电视剧中的珍珑棋局

一般的算法步骤都是经过小心设计的，并且深具目的性。然而随机算法恰恰反其道而行，在某些步骤中含有随机决定的指令。此举有如金庸小说中的虚竹，因为胡乱下一子后，竟然意外地解开了古今难破的珍珑棋局。随机算法的存在仿佛是在告诉人们，“道法自然”有时比“汲汲经营”有效率。



12.2 随机快速排序法

随机算法的第一个例子就是我们已经熟悉的快速排序法（可参阅第2章）。

如何避免快速排序法的最差情况？

让分割元素的值尽量靠近中间值。如此可以让分割更平均，并让快速排序法执行速度加快。

在执行分割前，先将矩阵的第一个数和其他任意数（由随机数决定）进行对调，就是一个极简单的随机快速排序法（randomized quicksort）。如此可大幅降低选中最大值或最小值为分割元素的机会，尤其是在数据输入时就可以大致排序好的情况。随机快速排序法如表12.1所示。

表 12.1 随机快速排序法

输入	$a[p], \cdots, a[q]$
输出	$a[p] \leqq a[p+1] \cdots \leqq a[q]$

步骤	Algorithm rquicksort(p, q)
	{
	if (p<q) then
	{if ((q-p) >5 then
	interchange (a, (Random() mod (q-p+1))+p, p);
	/*将矩阵a的第一个数和其他任意数进行对调*/
	j:= partition(a, p, q+1);
	rquicksort(p, j-1);
	rquicksort(j+1, q);
	}
	}

随机快速排序法的期望运行时间可被证明为 $O(n \log n)$，比原先的快速排序法的最差时间复杂度 $O(n^2)$快。

12.3 质数测试

第二个例子是质数测试(prime number testing)，如表12.2所示。

表 12.2 质数测试

问题	在通信安全的应用中常需要产生一个十分大的质数（除了 1 和本身外，没有其他正因子）。但是使用一般的方法测试一个正整数是否为质数有时不够快速。设计一个高效的算法来解决此问题
输入	正整数 n(>2) 13
输出	回答此正整数 n 是否为质数? 是

测试一个正整数 n 是否为质数的简单方法是，试着将 n 除以 2，3，…，$\lfloor\sqrt{n}\rfloor$。若没有数可以整除它，则 n 为质数；若存在一个数可以整除它，则 n 不是质数。这个方法看似不错，但是当 n 比较大时，计算非常冗长。

为了设计一个高效的质数测试算法，需要认识以下性质。

Fermat's Theorem

如果 p 是Fer，那么 $a^{p-1}=1(\text{mod}\ n)$，针对所有 $a\in\{1,2,\cdots,p-1\}$。

换言之，如果找到一个 $a\in\{1,2,\cdots,p-1\}$ 使得 $a^{p-1}\neq 1(\text{mod}\ n)$，那么 n 必不是质数。令人惊奇的是，当 $a^{p-1}=1(\text{mod}\ n)$时， n 为质数的概率也十分高。因此，这个条件（即 a^{p-1}是否等于 $1(\text{mod}\ n)$）就成为了检测质数的几乎完美的方法，如表12.3所示。

当随机质数测试算法判断输入值 n 不是质数时，一定是正确的。但是，当它判断输入值 n 是质数时，可以被证明错误概率不高于 2^{-s}。也就是当 s 足够大时，此算法的正确性是值得被期待的。

表 12.3 随机质数测试算法

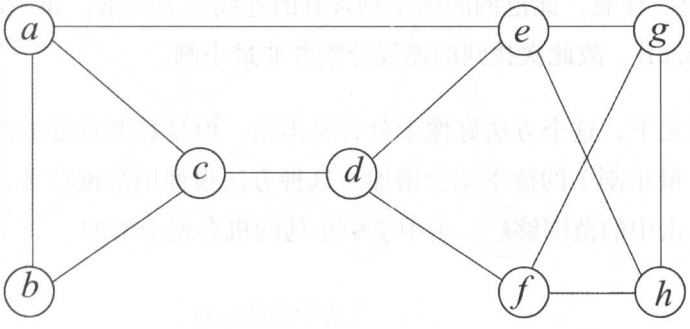
输入	正整数 n(>2)，正整数 s
输出	回答此正整数 n 是否为质数

步骤	Algorithm primality_test(n,s)
	{
	Step 1: for j=1 to s do　　//重复执行下列步骤 s 次
	{
	1.1: a=random(1, n-1);　　//从1 到 n-1的整数中随机选出一个数
	1.2: 如果 $a^{n-1}\neq 1 \pmod n$，那么返回n不是质数;　　//确定
	}
	Step 2: 返回 n 是质数;　　//不确定
	}

12.4 最小割算法

最后的例子是最小割算法 (minimum edge cut)，如表12.4所示。

表 12.4 最小割算法问题

问题	军方人员准备截断敌军某网络中的线路，以阻断其通信。为求时效，希望截断最少的网线分割整个网络，并使其断裂不连通。设计一个算法，找出这样的网线
输入	图 $G=(V, E)$代表现有的网络。每个顶点代表网络上的节点，而节点间的连线代表连接两个节点的网线 
输出	一个连线集合E的最小子集合 E'，使得除去此子集合E'后的图 $G'=(V, E-E')$是不连通的 $\{(a, e), (c, d)\}$

一个图上的连线分割（edge cut）为连线集合的子集合，若将此子集合移除，将导致此图不连通。最小割算法就是所有连线分割中拥有最少连线者。例如，在表12.4的图中，$\{(a, e), (c, d)\}$或$\{(b, a), (b, c)\}$就是一个最小割。

下面将设计一个随机算法来找出最小割。此方法十分简单，即随机重复地找到多组连线分割，并挑出其中最小的连线分割作为输出。

随机产生一个连线分割的方法十分简单。首先，任意（随机）从图中挑选一条连线，将此连线和其两端点挤压（contract）成一个点，但保留连接两端点的连线。图12.2显示了将连线 (c, d) 和其两端点挤压后的新图。

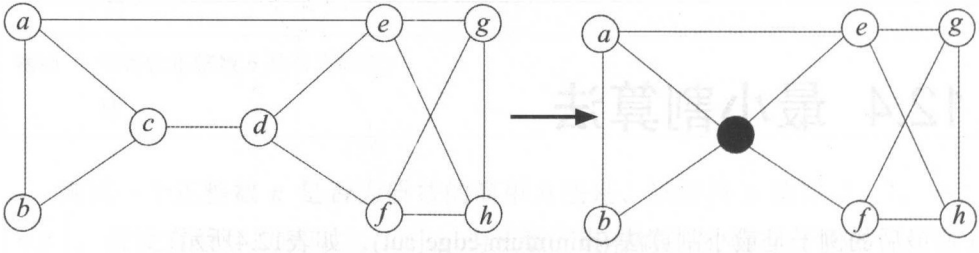
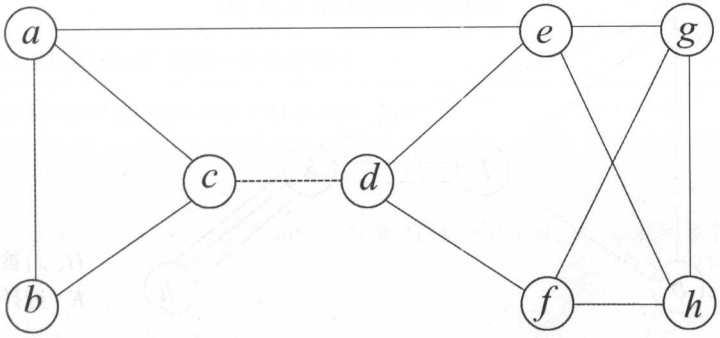


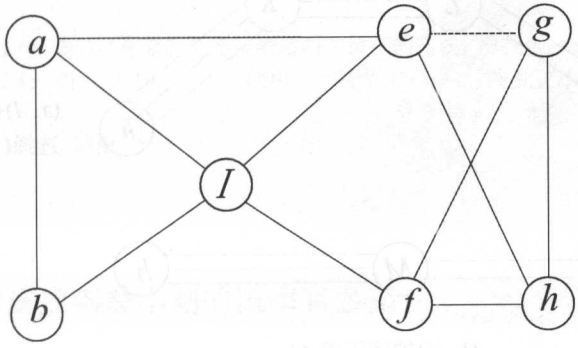
图12.2　挤压连线 (c, d) 和其两端点后，以一个新点取代此连线

重复以上操作，直到剩下两个顶点为止。注意，此时所有连接这两个顶点的连线就是一个连线分割（虽然有可能不是最小割）。例如，图12.3演示了找到一个连线分割的过程。倘若还原到输入图，所找到的连线分割是 {(e, h), (f, h), (g, h)}。注意，此范例的最小割含有的连线应为两条，如 {(a, e), (c, d)} 或 {(a, b), (b, c)}。故此次找到的连线分割并非最小割。

乍看之下，这个方法好像十分容易出错，但是只要重复的次数够多，找出正确解（最小割）的概率就会增加。这种方法很像用乱枪打鸟，只要射击的次数够多，击中的范围够广，击中空中小鸟的机会就会增加。

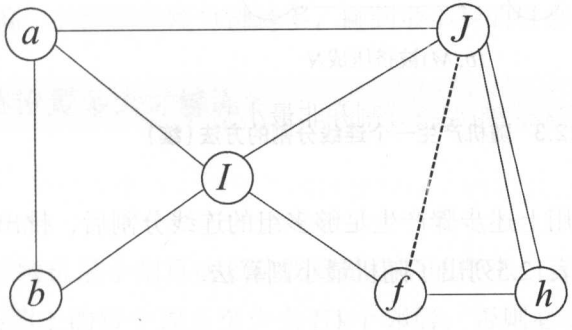


选择(c, d)



(c, d)被挤压成 I

选择(e, g)



(e, g)被挤压成 J

选择(f, J)

图12.3　随机产生一个连线分割的方法

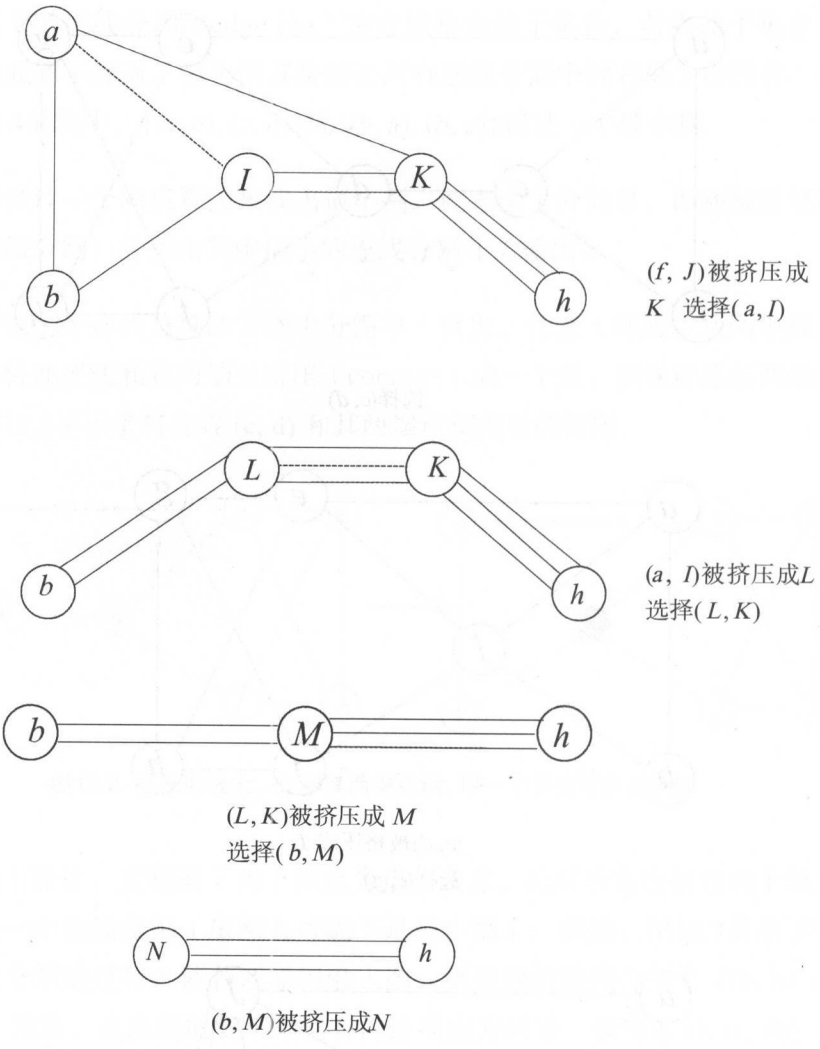


图12.3　随机产生一个连线分割的方法(续)

此随机算法是采用上述步骤产生足够多组的连线分割后，挑出其中最小的连线分割作为输出。表12.5列出了随机最小割算法。

表 12.5 随机最小割算法

输入	一个连接图 $G=(V, E)$及一个正整数 s
输出	最小割算法 C(正确率大于 $1-(1-2/n^2)^s$)

步骤	Algorithm minimum_edge_cut(G, s)
	{
	Step 1: for j=1 to s do　//重复执行下列步骤s次，以找到多个连线分割
	{
	1.1: V'=V; E'=E; C=E;
	1.2: 当 V'>2 则 do
	{
	随机地从连线集合 E'中选择一条连线 e=(u, v);
	挤压 e 成一个新的顶点，并且更新 V'和 E';
	}
	1.3: 令 C'存储所有连接最后两个顶点的连线;//找到一个新的连线分割
	1.4: 当|C'|<|C|，使用 C'取代 C;　//找到更小的连线分割
	}
	Step 2: 输出 C;
	}

这个算法找到最小割的概率有多高?

看起来不太高，但应该和s有关。

有什么关系?

s的值越高，找到的连线分割越多，碰到最小割的机会也就越大。

这样s该设置多大才够呢?

这个……

下面讨论随机最小割算法正确找到最小割的概率。首先，令最小割C共有k条连线，则图上的每个顶点至少会有k个邻居；否则必有一个顶点和其邻居的连线，从而形成一个比k少的连线分割（出现矛盾）。因此，此图至少拥有(nk)/2 条连线，此处n为图中顶点的个数。

希望计算出在随机选择连线（并挤压这条连线）的过程中，最小割C中的k

条线都不被选中的概率，即最后找到最小割的概率。注意，随机选择连线并挤压这条连线的操作共需要n-2 步，因为每执行一次操作（选择连线并挤压这条连线），剩下的图便少一个顶点，如图12.3所示。

因此，我们想计算在 n-2 次随机挤压过程中不会选中最小割C的概率。

第一次随机挤压的过程中，不会选中最小割C的概率为：

Pr[E_1]等于 1-(最小割C的连线条数)/(所有连线的条数) = 1 - k/(所有连线的条数)

因为此图至少拥有 (nk)/2 条连线，故：

$\Pr(E_1) \geqslant 1-(k/(nk/2))=1-2/n$

类似地，第二次随机挤压的过程中不会选中最小割C的概率为：

$\Pr(E_2) \geqslant 1-(k/((n-1)k/2))=1-2/(n-1)$

类似地，我们可得：

$\Pr(E_3) \geqslant 1-(k/((n-2)k/2))=1-2/(n-2)$

$\Pr(E_{n-2}) \geqslant 1-(k/((3)k/2))=1-2/(3)$

此处，E_i 代表在第 i 步时并未挑中最小割C中的连线，而 $\Pr(E_i)$代表其概率（$1 \leqslant i \leqslant n\text{-}2$）。

最后，在 n-2 次随机挤压（一条连线）的过程中，都不会选中最小割 C 的概率将大于或等于：

$$(1-2/n)\times(1-2/(n-1))\times\ldots\times(1-2/3)$$

$$=\frac{n-2}{n}\times\frac{n-3}{n-1}\times\ldots\times\frac{2}{4}\times\frac{1}{3}=2/(n\times(n-1))\geqslant 2/n^2$$

随机最小割算法执行上述操作 s 次，因此都得不到最小割 C 的概率将小于$(1-2/n^2)^s$，也就是得到最小割 C 的概率将大于 $1-(1-2/n^2)^s$。若选取较大的 s 值（如 $n^2/2$ 的倍数），则此算法出错的概率将降低到可以接受的程度。

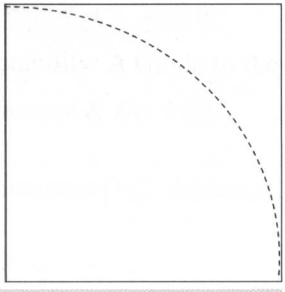
12.5 随机算法技巧

随机算法的优点在于简洁、高效。一般分成两类：拉斯维加斯算法（Las Vegas Algorithm）和蒙特卡罗算法（Monte Carlo Algorithm）。

拉斯维加斯算法总是输出正确的答案，随机的步骤影响的是执行的时间，如随机快速排序法。相对地，蒙特卡罗算法有时会输出错误的答案，但是独立地重复执行多次会将错误的概率降低，如随机质数测试算法和随机最小割算法。

学习效果评测

1. 设计一个蒙特卡罗算法来计算 1/4 圆的面积。在此正方形中均匀地产生大量的点后，利用落在 1/4 圆内点的比例计算此 1/4 圆的面积。



输入

```
10              (正方形的宽k)
```

输出

```
78.53           (1/4圆的面积)
```

2. 设计一个随机快速排序法，并与原来的快速排序法进行比较。

输入

```
5               (要排序的数字个数)
65              (以下是要排序的数字)
50
55
```

```
45

输出
45 50 55 65          (排好序的数列)
0.0012 秒            (排序所需的时间)
```

3. 质数比例：运用费马定理设计一个随机质数测试算法。尝试计算在任意连续整数的范围内有多少比例是质数。

```
输入
1 100                (m、n 代表连续整数的范围 m~n)

输出
25                   (在范围 m~n 中，所有质数的个数)
0.25                 (出现质数的比例)
```

参考文献

[1] Alfred V Aho, John E Hopcroft, Jeffrey D Ullman. The Design and Analysis of Computer Algorithms[M]. Addison-Wesley, 1974.

[2] J A Bondy, U S R Murty. Graph Theory with Applications[M]. North-Holland, 1976.

[3] T H Cormen, C E Leiserson, R L Rivest. Introduction to Algorithms[M]. The MIT Press, 1998.

[4] Shimon Even. Graph Algorithms[M]. Cambridge University Press, 2011.

[5] Lestor R Ford, D R Fulkerson. Flows in Networks[M]. Princeton University Press, 1962.

[6] Michael R Garey, David S Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness[M]. San Francisco: W. H. Freeman & Co, 1979.

[7] R L Graham, D E Knuth, O Patashnik. Concrete Mathematics[M]. Addison-Wesley, 1994.

[8] Dorit S Hochbaum. Approximation Algorithms for NP-hard Problems[M]. PWS Publishing Company, 1997.

[9] Ellis Horowitz, Sartaj Sahni. Fundamentals of Computer Algorithms[M]. Computer Science Press, 1978.

[10] Ellis Horowitz, Sartaj Sahni. Fundamentals of Data Structures[M]. Maryland: Computer Science Press, 1976.

[11] Eugene L Lawler. Combinatorial Optimization[M]. Holt Rinehart and Winston,

1976.

[12]R C T Lee, R C Chang, S S Tseng, Y T Tsai. Introduction to the Design and Analysis of Algorithms[M]. Flag Publishing, 2001.

[13]C L Liu. Introduction to Combinatorial Mathematics[M]. McGraw-Hill, 1968.

[14]Udi Manber. Introduction to Algorithms: A Creative Approach[M]. Addison-Wesley, 1989.

[15]G Ploya. How to Solve It[M]. Garden City, NY: Doubleday, 1957.

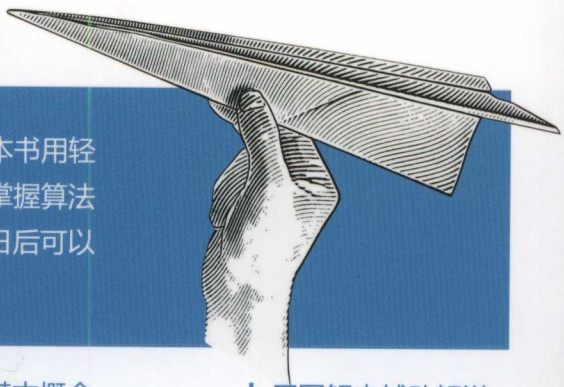
[16]Artaj Sahni. Concepts in Discrete Mathematics[M]. Camelot Pub Co, 1981.

[17]Robert Sedgewick. Algorithms[M], 2nd ed. Addison-Wesley, 1988.

[18]M N S Swamy, K Thulasiraman. Graphs, Networks and Algorithms[M]. John Wiley and Sons, Inc, 1981.

[19]Robert Endre Tarjan. Data Structures and Network Algorithms[M]. Society for Industrial and Applied Mathematics, 1983.

Algorithms



算法是利用计算机解决问题的技巧。本书用轻松的对话手法帮助读者简单且自然地掌握算法的基本概念，并养成“猜”的习惯，日后可以主动思考，尝试解决问题。

- 用轻松的对话手法，培养读者主动思考问题
- 注重基本概念，避免复杂的证明过程
- 用图解来辅助解说，减少冗长的程序代码

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/算法

ISBN 978-7-111-57887-1

9 787111 578871 >

定价：59.00元